AD-A265 866

RL-TR-93-28
In-House Report
February 1993

INITIAL DEFINITION OF A KNOWLEDGE-BASED SOFTWARE QUALITY ASSISTANT

Joseph A. Carozzoni

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-93-28 has been reviewed and is approved for publication.

APPROVED: Samuel A. DiNitto, Jr.

SAMUEL A. DINITTO, JR., Chief
Software Technology Division
Command, Control & Communications Directorate

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1993	In-House

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
INITIAL DEFINITION OF A KNOWLEDGE-BASED SOFTWARE QUALITY ASSISTANT	PE - 62702F PR - 5581 TA - 27 WU - 64
6. AUTHOR(S) Joseph Carozzoni	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Rome Laboratory (C3CA) 525 Brooks Road Griffiss AFB NY 13441-4505	RL-TR-93-28

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Rome Laboratory (C3CA) 525 Brooks Road Griffiss AFB NY 13441-4505	

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Joseph Carozzoni/C3CA (315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited.	

13. ABSTRACT (Maximum 200 words)

Prior and current research in the area of software quality reveals the necessity for a more innovative approach. This project suggests the need for further exploratory work in the application of knowledge-based and expert system technology to areas related to software quality. Prior investigation have focused on the development of an expert system to support the specification and generation of software quality factor goals. This project contemplates the creation and development of a comprehensive Knowledge-Based Software Quality Assistant (KBQA), which would eventually encompass the entire software development life cycle.

14. SUBJECT TERMS	15 NUMBER OF PAGES
Software Quality, Knowledge-Based	108
	16 PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Prior and current research in the area of software quality reveals the necessity for a more innovative approach. This project suggests the need for further exploratory work in the application of knowledge-based and expert system technology to areas related to software quality. Prior investigations have focused on the development of an expert system to support the specification and generation of software quality factor goals. This project contemplates the creation and development of a comprehensive Knowledge-Based software Quality Assistant (KBQA), which would eventually encompass the entire software development life cycle.

Software quality is commonly defined as the composition of thirteen software quality factors. The most quantifiable of the factors is reliability. Because of this, the initial work on the KBQA concept will be based on the RELIABILITY quality factor. Presently, software developers focus their attention on simply counting the faults/defects and failures found in a program. The bulk of this activity occurs in the latter stages of the software development life cycle, namely the System Test phase and the Operational and Maintenance phase. The more cost effective approach is to deal with reliability at the very beginning of the life cycle, and account for it right up until to the end of the life cycle. In addition, a knowledge-based system would be able to provide a feedback loop that allows for a continual improvement in the reliability engineering process.

In this project, the preliminary design of a KBQA is presented. The key to the success of this approach is based upon the development of the Reliability Component concept, with the three components of reliability being *prediction*, *estimation*, and *measurement*. This approach exploits the capability of a knowledge based expert system to be able to assist in the prediction and estimation of reliability in the earlier stages of the software development life cycle. The KBQA should be able to apply the correct proportion of estimation, prediction, and measurement of reliability at the appropriate stages of the software development life cycle. Several alternative knowledge base architectures which can support life cycle reliability concerns will be developed. In addition, the issues concerning the support of requirements-specific traceability and monitoring of reliability will be addressed. Hopefully, this approach would allow for automatic generation of a requirements-specific test plan.

CHAPTER ZERO

New Terminology

CHAPTER ZERO - NEW TERMINOLOGY

A key concept of this effort is to deal with reliability throughout the software development life cycle. This project treats the terms *prediction*, *estimation*, and *measurement* differently than conventional research does. The following table should provide a means to minimize confusion when reading this document:

Terminology	Conventional Meaning	Our Meaning
PREDICTION:	Normally the level of reliability which can be predicted by use of reliability models. Requires execution of the program.	Anticipated level of reliability prior to program execution. The *prediction* is based upon the software itself, and how the software was produced.
ESTIMATION:	Normally the level of reliability which can be estimated by use of reliability models. Requires execution of the program.	Anticipated level of reliability based upon a random sampling of program modules. The estimation is derived from statistical trend analysis techniques.
MEASUREMENT:	Actual reliability.	Same.

CHAPTER ONE

Introduction

CHAPTER ONE - INTRODUCTION

Significant progress has been made in the establishment of Software Engineering as an engineering discipline. Most notably is the development of computer-aided software engineering (CASE) tools for software development, management, and evaluation. Also, there exists a growing use of knowledge-based systems for software development, management, evaluation, and maintenance. As a consequence, there is the growing availability of improved life cycle support for software development. The two most prominent examples of attempting to address the life cycle support for software are the Department of Defense with DOD-STD-2167A and the Institute of Electrical and Electronic Engineers (IEEE) with P-1074.

In view of this, software development has not enjoyed the same status and success as has hardware development. In defense of software developers, the field is relatively young (in comparison to the other accepted engineering fields) and the state of the art is immature, at best. The most frequent criticisms (with the associated causes), often justifiable, are:

- Lack of corporate understanding of software development, thus leading to a lack of commitment to quality software engineering.
- Few individuals with an adequate level of software project management education and experience, thus leading to poor management of software developments.
- Inadequate employment of software engineering principles.
- Lack of software quality control (in comparison with hardware).
- Less than twenty universities with a software engineering curriculum (per June 1991 SEI report), thus leading to inadequate training in software engineering for both developers and project managers.

1.1 Why do we need software quality?

Society has understood the need for hardware quality from the very beginning. The simplicity of use of the "hardware" product is the primary reason. Either the bridge collapses under you or it doesn't, or either the airplane crashes or it doesn't. The results are easily traceable to something an individual can see, allowing self understanding and explanation. On the other hand, software quality has lagged in receiving adequate attention since the general public cannot "see" it. They can see the bridge, but not the CAD/CAM software that designed it. Newer models of airplanes have computer controlled software (fly by wire control). People can see and ride the plane, yet they not only do not see the software controlling the flight, most likely they do not even know that is the case! Did hardware win Desert Storm? The Iraq's had plenty of hardware.

The most convincing argument for software quality is the fact that it is all around us. We are continually increasing our dependence upon software-based systems, and the rate of dependence on it is accelerating. Coupled with this is the ever increasing costs of software failure. The economic consequences are no longer the grand worry, even though they are considerable. We are now predominately preoccupied with human-life consequences of software failure (i.e. safety critical applications).

There are many definitions for software quality, but they all ultimately relate to customer satisfaction (along with the goal of continued future business). The best proof of this is evident in the United States automotive industry. It was the effect on customer satisfaction by quality which ultimately led to their decline. Customer satisfaction is the primary reason for either a business's success or failure. The software development business has long enjoyed the following status:

- The common acceptance of execution failures as being "normal" for software (yet being unacceptable for hardware).
- The common acceptance of projects being late and/or over budget as being "normal" for software (yet being unacceptable for hardware).
- The common acceptance of the ever escalating cost of software-development as being "normal" for software (yet being unacceptable for hardware).
- The ability to excuse it as "software is different."

Above, we have alluded to the fact that the two primary justifications for placing a greater emphasis on software quality to be the ever increasing dependence on software-based systems, and the increasing costs of software failure. I have an even more important reason which I personally believe in. Beware the Japanese. In avoiding a religious argument, suffice it to say that the Japanese have slowly taken over many areas and are world leaders in those areas by meticulous attention to quality. Software, if not next, is in their pipeline.

1.2 Historical Perspective

Previous Rome Laboratory work in area of software quality defined a hierarchial model which identified thirteen software quality factors. The model hierarchy was based around the idea of software quality factors, criteria, and metrics. The framework was defined as:

1. Software Quality Factors - Behavioral characteristics of the system.
2. Software Quality Criteria - Decomposition of the factors into attributes which relate to software itself.
3. Software Quality Metrics - Measurements of the criteria.

The basis for the concept of employing software quality factors has roots in RADC-TR-77-368, titled Factors in Software Quality by McCall, et al. This effort had a follow-on effort to further define the work in RADC-TR-85-37, itled Specification of Software Quality Attributes by Bowen et al, from the Boeing Company. This effort resulted in the RL (formally RADC) Software Quality Framework, a three volume set detailing the specification and evaluation of software quality in a manner consistent with Department of Defense standards. Volume I was a general introduction to software quality attributes. Volume II was the specification guidebook which dealt with the specification of software quality goals, tailoring of the framework for individual applications, and the prioritization of the software quality factors. Volume III was the evaluation guidebook and addressed the evaluation of the software quality factors and measurement of the achieved product.

The representation of software quality as defined by the RL Software Quality Framework is categorized by the three acquisition concerns of performance, adaptation, and design. The thirteen software quality factors grouped into their respective categories are:

Performance:
- Efficiency
- Integrity
- Survivability
- Usability
- Reliability

Adaptation:
- Portability
- Reusability

Expandability
Flexibility
Interoperability

Design:
Correctness
Maintainability
Verifiability

In an attempt to assist software developers in use of the framework, three CASE tools had been developed. One tool, The Assistant for the Specifying of the Quality of Software (ASQS) was designed to assist in setting software quality requirements and goals, and to tailor the framework for individual projects. Another tool, The Quality Evaluation System (QUES) was developed to automate the evaluation guidebook of the software quality framework. Lastly, the Automated Measurement System (AMS) is a tool for supporting quality analysis. With an input consisting of the requirement specification, preliminary design, detailed design, code, and software problem reports, it provides an output of management reports, quality reports, and statistical reports. These tools were develop with the goal of interfacing to the Software Life Cycle Support Environment (SLCSE). In general, these CASE tools have not provided significant encouragement for software developers to more widely and thoroughly use the RL Software Quality Framework. The manner in which these tools were to be employed were as follows:

- First, users in consultation with the software developers would utilize the ASQS to identify the desired *Software Quality Specification*.
- Software developers would then utilize the QUES to perform a software *Quality Measurement*.
- Software developers would then inform the users of the results using the AMS in *Evaluation Reports*.

Much has been written of the RL Software Quality Framework, both in favor and in opposition. However, both sides agree on four key points. First, the Framework works! But... Second, the Framework is considered to be too complex and expensive to implement. Third, the Framework does not provide for a smoothly integrated life cycle approach for dealing with software quality issues, in that it is biased on initial specification of quality, then jumps to final measurement. Much of the software development process in the intermediate stages is left out. Fourth, the Framework is more subjective than objective.

Present day thinking is on the necessity of engineering in the quality right from the very beginning. The disagreement is over the way in which quality will be achieved, especially in an affordable and practical manner. Software is sufficiently different from hardware such that emulating the hardware way of doing business is inadequate. This is most unfortunate because the hardware industry has a long and successful history of quality success, and much of their pioneering work could have been exploited with minimal effort.

In order to better understand what has the greatest effect on quality, many other factors must be looked at. Some of the factors which have a tremendous effect on software quality, but have had less attention are those more involved in the actual management of the software development process, such as:

- The quality of requirements analysis.
- The quality of requirements specification.
- The development process.
- The development method.
- The development environment/CASE.
- The quality of the developer.
- The schedule/budget.

1.3 Zeroing in on Reliability

As stated above, software quality is commonly defined as the composition of thirteen software quality factors. All of these factors are important and should be appropriately addressed. Of the thirteen software quality factors, the only ones that have well established procedures in place for quantifying of are the factors maintainability and reliability. It is generally assumed that of these, reliability is the most important. Because of this, the initial work on the Knowledge Based software Quality Assistant (KBQA) concept will be focused upon the RELIABILITY quality factor.

There are many factors which contribute to the reliability of a software system. Factors of interest include different development scenarios and processes, different testing strategies, differences in the skills and experience of the developers, to name a few. It is easily shown that software reliability, and quality in general, correlates with various known and studied factors, but accurately calculating the reliability from these factors seems impossible, or at least difficult. In the present state of software reliability engineering, most are simply content with handling reliability as a function of the amount of effort and money spent on testing. Studies confirm this conception, but at what cost?

In this project, the preliminary design of a KBQA is presented. The key to the success of this approach is based upon the development of the Reliability Component concept, with the three components of reliability being prediction, estimation, and measurement. This approach exploits the capability of a knowledge based expert system to be able to assist in the prediction and estimation of reliability in the earlier stages of the software development life cycle. The KBQA should be able to apply the correct proportion of estimation, prediction, and measurement of reliability at the appropriate stages of the software development life cycle. Several alternative knowledge base architectures which can support life cycle reliability concerns will be developed. In addition, the support of requirements-specific traceability and monitoring of reliability will be addressed, as will the automatic generation of a requirements-specific test plan.

CHAPTER TWO

Problem Description

CHAPTER TWO - PROBLEM DESCRIPTION

2.1 Introduction

Of all the software quality factors in common use, only reliability and maintainability have a well defined measurement and evaluation procedure in place. Of these two, the reliability factor is more important and has been better defined. Too often, it is assumed that reliability quantification can only be used in the latter stages of software development. However, techniques exists that could allow us to involve reliability evaluation in all phases of the life cycle. Measurement of reliability is currently practiced mostly in the System Testing phase and the Operational and Maintenance phase of the life cycle. In certain situations, some researchers have attempted to estimate expected patterns of reliability using partial system execution. This is normally attempted in those phases that involve some form of module execution of the software (i.e. Unit Testing). Obviously, before any code has been written, we cannot measure (or estimate) the reliability of the software. However, we are capable of developing procedures for the prediction of reliability in earlier phases of the life cycle such as the design phase. It is even desirable to perform reliability engineering in even earlier phases. This hints at the possible development of a process for handling software reliability throughout the software development life cycle. This thinking, called reliability engineering, is considered to be the most cost effective way of developing software for reliable operation (or quality in general).

A common definition is: " Software reliability engineering is the applied science of predicting, measuring, and managing the reliability of software-based systems to maximize customer satisfaction." In spite of this accepted definition, the practice of prediction is not being applied, and the practice of estimation is not much more than a mirage. There are many reasons favoring the application of software reliability appraisal throughout the software development cycle. Some of the most important are:

- The prediction of reliability during conceptual phases.
- The establishment of realistic numerical reliability goals during definition phases.
- The establishment of existing levels of achieved reliability.
- The monitoring of progress toward achieving specified reliability goals or requirements.
- The establishment of reliability criteria for formal qualification.

Electronic engineers have been successfully applying reliability engineering techniques to hardware systems for quite some time now. Using several ideas borrowed from the hardware development arena, and creating some new concepts, Musa defines software-reliability engineering as:

- Helping select the mix of principal quality factors that maximize customer satisfaction.
- Establishing an operational profile for the system's functions (frequency of function use).
- Guiding selection of product architecture and efficient design of the development process to meet the reliability objective.
- Predicting reliability from the characteristics of both the product and the development process.
- Estimating reliability from failure data in test, based on models and expected use.
- Managing the development process to meet the reliability objective.
- Measuring reliability in operation.
- Managing the effects of software modification on customer operation with reliability measures.
- Using reliability measures to guide development process improvement.
- Using reliability measures to guide software acquisition.

The initial definition for a Knowledge Based software Quality Assistant (KBQA) will be centered around reliability. Let it be noted that the other software quality factors have not been forgotten with this initial focus on reliability. This "software-first" life cycle approach developed for reliability will be designed to be extensible to the other software quality factors in the future.

2.2 Present Practice

Software reliability in an operational environment can presently be accurately measured. This is where the software failure intensity models are normally employed. Often, it is normally straight forward to emulate the operational environment when performing testing. Some general reliability terminology (IEEE) for discussing reliability related matters are:

Failure - departure during operation

- A property of execution behavior.
- Can only be observed when the program is executing.

Fault - defect in the program which when executing causes the failure.
- A property of the program itself.
- Can be observed when the program is not executing.

The premier body of standards development in this field is the IEEE. They have published an "Unapproved Draft - Published for Comment Only" document titled Standards for a Software Quality Metrics Methodology, dated April 1, 1990. The standard is essentially a clone of the Rome Laboratory (RL - formally RADC) Software Quality Framework. In section 3.0 of the document, titled Purpose of Software Quality Metrics, the use of metrics is heavily weighted to the specification and measurement of software quality. The reliability thinking is augmented toward the behavior of failures, which are affected by two principal factors:

1. The number of faults in the software being executed.
   - difference between the number introduced and the number removed.
2. The execution environment or operational profile of execution.

This emphasis on execution behavior is unfortunate since at that late time, it is "after-the-fact". Earlier life cycle reliability concerns is centered around the occurrence of faults. A few fault analysis techniques exist and are normally weighted toward code size and code complexity metrics. However, even at this point, considerable time and expenditures have been expended on the earlier phases of the life cycle. There is very little emphasis placed on early life cycle reliability engineering, and what little there is, has not been defined or developed into a manner which encourages use (or even is usable).

Reliability models have usually been defined with respect to time, although it would be possible to define them with respect to other variables. Other metrics may include the breadth of testing, the depth of testing, the number of test cases run, and the percentage of requirements tested. The two primary kinds of time that are considered are:

1. Execution Time - is more important to the developer.
2. Calendar Time - is more important to the customer.

Several reliability models are well developed, and have been in wide use during the System Test and Operational and Maintenance phases of the life cycle. The four general ways of characterizing failure occurrences in time are:

1. The time of the failure.
2. The time interval between the failures.
3. The cumulative failures.
4. The failures experienced in a time interval.

In summary, the present techniques employed to **measure** reliability are well established and seem to work. The present techniques to **estimate** reliability are based upon partial program execution techniques. Their development is incomplete and the accuracy of their result are limited. Estimation techniques have seen use, but to a lesser extent. The present techniques to **predict** reliability are not well founded, thus they have not been used. Also lacking is a smoothly integrated life cycle approach to manage reliability engineering. In general, this is also true for the other factors representing software quality.

2.3 A Better Way

To meet the software quality challenge, reliability considerations must be pushed up to the front of the life cycle. Also, well defined procedures must be used to monitor and track the progress of meeting the desired reliability goals. During the software development process, we must be able to move away from relying on post-coding measurement, and start to develop a RELIABILITY PREDICTION capability. Also, we must develop a better procedure such that RELIABILITY ESTIMATION is more frequently and accurately employed. Finally, an automated tool to assist in choosing the best reliability measurement model must be developed to assist software developers during the later stages of the life cycle.

Some of the ingredients used to predict reliability are more common sense than science. The foundation for reliability prediction can be based upon two primary criteria:

1. Software Domain: inherent problems (i.e. real-time embedded systems).
2. Software Development Process: i.e. SEI Software Capability Maturity Measurement.

We can also perform pseudo-measurement (still considered to be only a prediction) of reliability by an analysis of the practices in use by the software developers. Such traits to look for are:

- Proper educational/training programs in place.
- Lead developers already at a high level of capability/experience.
- Established testing procedures in place.
- Successful history of continually increasing reliability.
- Proper management process in use.

Next, we must be able to estimate reliability after coding has been initiation. Using advanced statistical trend analysis techniques, we should be able to employ reliability *estimation* procedures. This involves accurate parameter estimation using "partial system" (partial execution/testing) data.

Finally, optimizing the *measurement* (i.e. testing/V&V) process is another high priority of the KBQA. Reliability generally increases with the amount of testing, but it can also increase with optimizing the choice of test procedure selection and test case selection and generation. Thus reliability can be closely linked with project schedules, and intimately tied in with project management. Some desired features/capabilities include:

- Assistance in choosing the best reliability model for individual models/systems.
- Automatic test plan generation.
- Testing left to reach goals.
- Testing/cost trade-offs.

An aspect of software development that must also be addressed concerns software reuse. The general beliefs which seem to favor a greater emphasis on the practice of software reuse are:

1. In general, only code that is new or has been modified results in fault introduction. Code that is reused from other applications does not usually introduce any appreciable number of faults, except in the interfaces.

2. Code which is reused is normally debugged in other applications. The only possible consideration is reuse of code that has a different "level" of reliability.

In summary, the present practice is to place far too little emphasis upon the use of prediction in software reliability engineering. Also, a greater emphasis must be place on the use of estimation in software reliability engineering. The bulk of the reliability related emphasis is placed on post coding measurement using several established reliability models (i.e. measurement of execution behavior). At this point in the life cycle, it is much more expensive to correct reliability deficiencies. This practice is unfortunate, since only after the system has been "delivered" to the customer, does concern develop as to whether or not the initial reliability goals have been meet. Should the reliability goals not be attained, it is much too late to significantly impact what could have been done.

2.4 Project Scope

As stated above, the present emphasis on the quantification of reliability is based towards the end of the life cycle where actual reliability data is available (i.e. failures per cpu hour). The scope of this effort is to develop the initial definition of a Knowledge Based software Quality Assistant (KBQA) with a focus on reliability. The KBQA will employ a three pronged attack to address the software reliability issues:

1. Reliability Prediction - determination from properties of the software product and the development process (prior to any execution of the program).

2. Reliability Estimation - statistical inference procedures are applied to failure data taken from partial program execution (i.e. unit execution).

3. Reliability Measurement - use of an expert system to assist in optimizing the choice of reliability models to employ based upon the prevailing circumstances. Included with this is the optimization of test procedure selection and test case selection and generation.

A key capability of the KBQA will be to "blend" the three (prediction, estimation, and measurement) into a single threaded process. Previously, this has not been possible. However, use of knowledge based expert systems will provide the capability to implement this process.

CHAPTER THREE

Reliability and the Life Cycle

CHAPTER THREE - RELIABILITY AND THE LIFE CYCLE

3.1 Introduction

The intent of using a life cycle approach to software development is threefold. First, to improve the efficiency of the software development process. Second, to provide an avenue for efficient maintenance of deployed software. Third and most importantly, is meeting the user requirements. There is a variety of software development models that address the life cycle approach, each of which have been developed to meet a particular users needs. Regardless of the number of different life cycle models in existence and use, reliability engineering can be applied to any of them. The most important business that must be conducted is the application of reliability engineering throughout the life cycle, for which ever one is used.

3.2 DOD-STD-2167

For this project, the task can be simplified by focusing discussion on a single life cycle. The life cycle that will be used to discuss my proposed life cycle approach to software reliability engineering will be a simplified DOD-STD-2167A life cycle. My academic sponsor for this project is the United States Air Force, thus I am proficient with this life cycle. From the document DOD-STD-2167A, titled Defense System Software Development dated 29 February 1988, the military standard for the life cycle is defined as:

a. System Requirements Analysis/Design.
b. Software Requirements Analysis.
c. Preliminary Design.
d. Detailed Design.
e. Coding and CSU Testing.
f. CSC Integration and Testing.
g. CSCI Testing.
h. System Integration and Testing.

The system is considered to be composed of several components; Computer System Unit (CSU), Computer System Component (CSC), Computer System Configuration Item (CSCI), System Segment, and System. A hierarchial diagram of this breakdown is as follows:

Unit (CSU) --> Component (CSC) --> Configuration Item (CSCI) --> Segment --> System

The DOD-STD-2167 military standard has just recently undergone a revision, and the CSCI category has been removed in an attempt to streamline the standard. The hierarchial diagram of this new breakdown is as follows:

Unit (CSU) ----> Component (CSC) ----> Segment ----> System

It is likely that DOD-STD-2167A will follow suit in 1993 when it is up for revision. Thus the definition of a software system will probably have a nomenclature consisting of units, components, segments, and systems.

3.3 Generic Life Cycle

I have made some minor modifications to the DOD-STD-2167A life cycle model by simplifying it to make it clearer for our purposes. The intent here is to focus on reliability engineering, and not life cycle religion. The modifications include the addition of a "Feasibility" stage and an "Operational and Maintenance Phase". The deletions are centered around the system/sub-system and segment partitions. The life cycle that we will concentrate on looks like this:

0. Feasibility Phase.
1. Requirements Phase.
2. Design Phase.
3. Coding/Unit Testing Phase.
4. Integration/Integration Testing Phase.
5. System Testing Phase.
6. Operational and Maintenance Phase.

The Feasibility phase is indexed with a zero to clarify the fact that it is not an actual phase of traditional software development. However, even as a precursor to software development, it is a phase where reliability can be addressed. The Operational and Maintenance phase is also not ordinarily considered to be part of the software development process. However, in focusing in on reliability, much can be learned about how the system development process affects software reliability by studying system behavior in the operational environment. It is hoped that by using some form of "feedback-loop", that lessons can be learned about software reliability and then be used to refine the reliability engineering of software development. Only in the operational environment can we definitively measure what we have accomplished.

3.4 The Reliability Component Concept

Reliability can be handled throughout the life cycle by three means: Prediction, Estimation, and Measurement. It is not a clear division, however, and requires that the three techniques be blended into a single threaded process. In figure 3-1, a graph of the life cycle shows an example blending of prediction, estimation, and measurement based upon the state of the life cycle. A simple formula can be developed which we will base the reliability appraisal on:

Reliability := SUM (%Prediction + %Estimation + %Measurement)

Where the sum of the %'s must equal one. At any stage of the life cycle, the reliability of a system can be specified by the sum of the reliability "components" with their corresponding weights. In figure 3-1, it is easily seen that at the project's Requirements phase, the only component of reliability is that of Prediction. In other words, we can say:

Reliability := 100%Prediction + 0%Estimation + 0%Measurement

or

Reliability := 100%Prediction

Of course, this is more or less obvious, but it serves the point of illustration. At the other extreme of the life cycle, it is just as obvious that reliability in the Operational and Maintenance phase can be defined as:

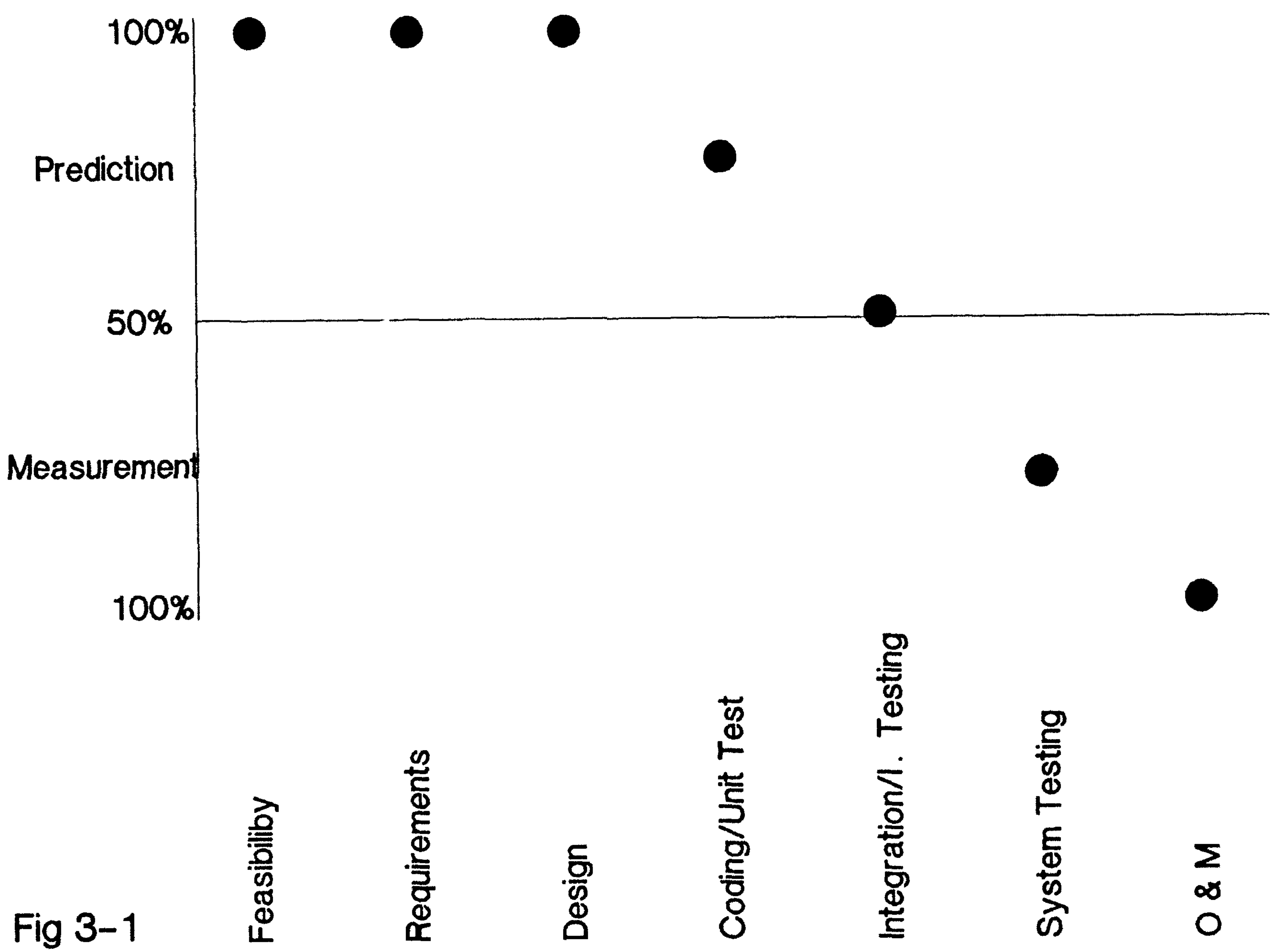

100%
Prediction
50%
Measurement
100%
Feasibiliby
Requirements
Design
Coding/Unit Test
Integration/I. Testing
System Testing
O & M

Fig 3-1

$$Reliability := 0\%Prediction + 0\%Estimation + 100\%Measurement$$

or

$$Reliability := 100\%Measurement$$

Where the formula becomes valuable is in the central area of the life cycle where reliability must be defined as a factor of all three components.

> Special Note: It should be noted in figure 3-1 that the reliability component mixture percentages are not hard and will vary from project to project. The figure should be used only as an illustrative representation of the reliability component concept.

Using this arbitrary example, it can be seen that the reliability in the Coding/Unit Testing Phase could very well be defined as:

$$Reliability := 20\%Prediction + 60\%Estimation + 20\%Measurement$$

The 20% prediction and the 60% estimation is intuitive. For estimation, the small sample size of units that have been tested can provide a valuable insight into the system's reliability using statistical trend analysis techniques. For prediction, there will probably still be a number of units not yet coded, thus for this portion of the system, we still must rely on the original predictions. Also, the combinatorial effects of the reliability of individual modules, once integrated, will have to be predicted. Then again, as values of the effects of integration on reliability becomes available, the estimation of these integration affects on reliability can be calculated - and fed into the equation. What may not seem intuitive is the 20% measurement - since measurement is normally considered to be possible only after coding and integration is complete. Using the DOD-STD-2167 nomenclature, there are segments that make up systems; i.e. sub-systems in their own right. In small systems, there will not be a measurement component and reliability will be given just by prediction and estimation. However, in places where a large system might be composed of, say, four independently running systems, and each independent of one another, we need a way to specify that some part of the system is indeed measurable. This may complicate things and cause some confusion, but I believe the added flexibility to account for such situations is well worth the effort.

3.5 Reliability: Phase by Phase

In handling reliability in a life cycle manner, we have to address what considerations must be accounted for in each phase of the life cycle. This phase by phase (i.e. serial) discussion of reliability is not meant to imply that the methodology will not work in another software development paradigm (i.e. Object-Oriented/Parallel). It is just that for our discussion, it simplifies the rationalization process of the software Reliability Component concept. Below is a table in bullet format of our example life cycle, with the methodology and considerations which are necessary to appraise reliability at each distinct phase.

0. Feasibility Phase: Pure Prediction.

Method: Very high level evaluation.

- Software development process (reference SEI).
- Application domain.
- Technological considerations (i.e. pushing state-of-the-art).
- Look at prior work in similar areas (success versus failure).

Goals:

- Establish reliability requirements.
- Perform high level tradeoffs.
- Relate reliability to user.
- Set reliability goals for system.
- Allocate reliability goals to hardware and software.
- System reliability assessment.

Common Question: "Is it worth it at this phase to attempt to model reliability predictions with so little information."

The Answer:

"The application of software reliability engineering techniques during this phase forces the developers to focus very early on reliability issues and baseline assumptions about the product's reliability **before** development begins. The situation would be no worse than if you didn't attempt to do it anyway."

1. Requirements Phase: Pure Prediction.

Method: Similar to Feasibility phase, but more detailed.

- Perform analysis of failure intensity objectives.
    - What does the customer consider to be a failure.
    - Severity level of each identified failure.
    - Estimated cost to achieve reliability goals.
    - Perform more detailed trade off analysis.
- Develop a prediction of the reliability based upon the operational profile/domain.
    - i.e. (AI-based software usually has reliability problems since V&V of this domain is still at it's infancy; however, mathematical domains such as signal processing are straight forward).
- Software development process (reference SEI).
- Technological considerations (i.e. pushing state-of-the-art).
- Look at prior work in similar areas (success versus failure).

Goals:

- Generate product-requirements specification (with reliability allocation).
- Analyze testability of requirements
- Analyze feasibility of requirements

2. Design Phase: Predictive in Nature/Estimative Potential To Be Determined.

Method: Similar to Requirements phase, but much more detailed.

- Transform the requirements specification into a design specification.
- Prediction based upon SEI CMM of the developers.
- Prediction must also address what is "technology feasibility".
    - i.e. to what level is the state-of-the art being pushed.
- Effects of software reuse and prior experience in this area.

Goals:

- Decompose and budget reliability requirements to software components.
- Establish design practices to encourage reliable software design.
- Analyze/simulate reliability performance.
- Predict software reliability.

3. Coding/Unit Testing Phase: Predictive and Estimative.

Method: Mixture of Design phase prediction with a first look at the actual fault density of random modules.

- Does the testing of selected units imply that the desired reliability can be achieved with the proposed design?
- Predict combinatorial effects of future unit integration.
- Estimation based upon partial execution (i.e. unit testing).
    - Can only achieve relative reliability measurements.
    - Statistical trend analysis techniques (i.e. connect the dots).

Goals:

- Build in reliability.
- Establish coding standards to encourage reliable software production.
- Conduct UT/Debugging to remove module level faults.
- Prototype builds for user feedback.
- Estimate software reliability.

4. Integration/Integration Testing Phase: Strongly Estimative.

Method: Unit Test Prediction of effects of Integration replaced with actual observations.

- Estimate complete system after integration.
- Predict changes due to laboratory and operational environment testing differences.

Goals:

- Test to requirements
- Test thoroughness evaluation
- Maintain standards during rework
- Insure test quality
- Regression testing
- Estimate software reliability
- Problem report statistics
- Acceptance testing

5. System Test Phase: Measurement with Some Prediction & Estimation.

Method: Estimative accuracy of laboratory environment.

- Predict and Estimate the measurement accuracy affected by the realistic emulation of the operational environment.

Goals:

- Hardware/Software Error Analysis
- Hardware/Software Reliability Integration
- System test thoroughness evaluation
- Insure test quality
- Regression testing
- Estimate system reliability
- Test assessment in operational environment.

6. Operational & Maintenance (O&M) Phase: Pure Measurement.

Method: Measurement of customer observations.

- Results can be collected and fed back into the process to assist in future predictions.

Goals:

- Regression testing
- Quality assurance
- Reliability measurement
- Fix existing faults.
- Enhancement.
- Trace reliability looking for long-term-usage degradation.

CHAPTER FOUR

Reliability Measurement

CHAPTER 4 - RELIABILITY MEASUREMENT

4.1 Introduction

In the next three chapters, the prediction, estimation, and measurement of reliability will be covered. My preferred order in which to deal with reliability is to start with the basic concepts of reliability measurement, then go on to reliability estimation, and then finally reliability prediction. Notice that this sequence is in the reverse order in which one would actually perform reliability engineering. Considerable work has been performed in the area of reliability measurement and has resulted in established techniques (and formulas) for measuring reliability. Reliability estimation will follow in chapter five. The proposed estimation techniques are based upon a partial set of reliability measurements, combined with sophisticated formulas which combine the reliability measurement models with trend analysis statistics. The goal is to estimate the final product's reliability based upon a partial execution of it's subcomponents. This area has seen limited use and understanding, but offers great potential. Finally in chapter six, reliability prediction will complete the discussion of the three components of reliability. This is largely an unexplored area, and offers great potential for further research.

4.2 Reliability Measurement: A Function of Testing

The generally accepted definition of software reliability is the probability of failure-free operation of a computer system in a specified environment for a specified time. Even though reliability is affected by the development environment and the design methodologies used on the project, these factors are not easily varied while the project is underway. Making major changes to either of those two factors is considered detrimental from a project management point of view (cost, schedule, and performance). This ultimately leads to depending upon Validation and Verification Techniques (i.e. Software Testing) as the primary determinate of reliability. The time required, resources needed, and costs associated with testing are mainly dependent on the initial reliability of the design and code and the reliability goal to be attainted.

The manner in which reliability is measured is by testing the software. There are predominantly two distinct approaches to software testing, with a third being a combination of the two. Following is a partial list of the more common software testing techniques:

1. **Structural (White/Glass Box)**
    - code reviews/inspections
    - statement
    - branch
    - multiple condition
    - domain
    - mutation

2. **Functional (Black Box)**
    - equivalence partitioning
    - boundary value
    - performance
    - random

3. **Hybrid (combination of Structural and Functional)**
    - *revealing sub-domains*

The final determinate of reliability measurement must occur in the operational environment. Frequently, during system testing, the operational environment is not available and must be simulated. Test data must then be injected into the various reliability models. Several models have been developed to support reliability measurement. Since the purpose of this project is to develop a Knowledge Based software Quality Assistant (KBQA), a detailed discussion of the theory of the reliability models will not be discussed. The only "theory" that is needed for this initial definition of the KBQA is:

- The models make the assumption that failures are independent of each other.
- The failures are a result of two processes:
    - introduction of faults.
    - their activation through selection of the input states.

The majority of the reliability models are classified as "reliability growth models". Their intent is to estimate the current reliability and to predict the future reliability growth for the software. The models base their measurement and predictions on only past failure times. There are three limitations to the models:

- The models treat software as a black box.
    - they do not incorporate information on program size, complexity, and other known metrics.
- The models do not make use of additional data which arises during testing.
- The approach does not give useful estimates for extremely high levels of reliability.

Special Note: The "prediction" offered by these reliability models are not to be confused with my definition of prediction. By prediction, I am referring to factors primarily associated with the software itself (i.e. size and complexity) and of the software development process (i.e. SEI Capability Maturity Measurement).

In summary, the various factors affecting reliability measurement are time between failures, number of runs between failures, source code metrics, number of discovered errors, programmers experience, testing method, time spent on testing, stability of the specification, testing compression, program category, and the level of programming technologies (design). One of the bigger issues, which is also a sleeper issue, which is very influential in affecting the level of reliability is the issue of software reuse! The KBQA could have the capability of justifying the cost of designing software for reuse. A trade off analysis showing the cost savings of reliable, reused software modules (which after a few projects should be extremely reliable), versus creating new software and testing it to the same level of reliability.

4.3 Software Reliability Model Classification

Following is a general classification of software reliability models with some of the more common models listed. Again, these models are based upon observed failure data, which implies execution of the program. The classifications of the models are partitioned into four categories. The categories along with some characteristics are as follows:

1. Time Between Failures

- Models postulate the shape of the error detection rate after each observed failure.

  Jelinski and Moranda De-Eutrophication Model
  Geometric De-Eutrophication Model
  Schick and Wolverton Linear Model
  Schick and Wolverton Parabolic Model
  Goel and Okumoto Imperfect Debugging Model

Hybrid Geometric Poisson
Littlewood-Verral Bayesian

2. Failure Count

- Models postulate the failure rate over a period (e.g. testing period), during which many errors could be detected.

Goel-Okumoto Non-homogenous Poisson process model
Schneidwind Model
Goel-Modified Non-Homogeneous Poisson Process Model
Musa Basic Execution Time Model
Musa Logarithmic Poisson Execution Time Model
Shooman Exponential Model
Generalized Poisson Model
IBM Binomial Model
IBM Poisson Model
Modified Jelinski and Moranda
Modified Geometric De-Eutrophication Model
Modified Schick and Wolverton

3. Input Domain Based

- The input domain is partitioned, and test points are chosen from each partition.

Nelson
Brown-Lipow
HO
Ramamcorthy and Bastani

4. Error Seeding

- Intentional errors are randomly seeded among the unintentional errors. Testing finds the indigenous and the seeded errors. This technique supposedly allows for accurate estimation off indigenous errors.

Mills Hypergeometric Model
Lipow's Extension of Mills Model
Basin's Extension of Mills/Libow Model

The intent of this project is not to provide a detailed examination of the different characteristics and behavior of each software reliability model. The immediate intention is to provide a technique supporting the optimization of model selection. Summarizing the key assumptions by model category:

Times Between Failures (TBF Models):

- independent times between failures
- equal probability of the exposure of each fault
- embedded faults are independent of each other
- faults are removed after each occurrence
- no new faults introduced during correction (i.e. perfect fault removal)

Fault Count (FC) Models:

- testing intervals are independent of each other
- testing during intervals is reasonably homogeneous (probability distribution does not vary with time).
- numbers of faults detected during non-overlapping intervals are independent of each other

Input Domain Based (DB) Models:

- input profile distribution is known
- random testing is used
- input domain can be partitioned into equivalent classes

Fault Seeding (FS) Models:

- seeded faults are randomly distributed in the program
- indigenous and seeded faults have equal probabilities of being detected

4.4 Expert System Based Model Selection

The first step in developing an architecture for optimal reliability model selection is determination of just what information the KBQA must provide. It is possible to create several decision tables which have the software reliability models on one axis, and the following choices on the other axis:

- What they can predict (e.g. MTTF, number of errors).
- What type of data they need as required parameters (e.g. time between failures, number of failures between observations, etc).
- What assumptions they are based on.
- What types of software they can analyze.
- Model appropriateness throughout the life cycle.

- etc.

The decision table approach helps in the development of a "cook book" style of reliability model selection. This still falls short in providing a true solution. With all the different models, the different phases of the life cycle, etc., there would be far too much data to wade through in a reasonable amount of time with any hope of optimizing the model selection. The decision table is a good starting point, but a more substantial automated methodology should be used to assist in the model selection.

The result of extending the idea of a decision table is the concept of a Decision Support System (DSS). Most of the work in the DSS area has been a combination of Operations Research and Artificial Intelligence (specifically Knowledge-based Expert Systems). The Operations Research approach works well in situations where mathematical models or "rigid" decision tables can be formulated. Operations Research falls short when the decisions lack structure and uncertainty exists amongst the decisions. Expert Systems, on the other hand, thrive in research areas where formal structure of decision is lacking, and uncertainty amongst the decisions exists. It would appear that selection of the appropriate software reliability model is a problem well suited for attacking by Expert Systems. This is especially true in later chapters of this project when reliability prediction and reliability estimation is addressed.

Use of these software models require that the program (or sub-programs) be executed. The applicable phases of the life cycle in which these software reliability models would most likely be employed are in the later stages of the software development life cycle, and are:

- Coding and Unit Testing Phase.
- Integration and Integration Testing Phase.
- System Testing Phase.
- Operational and Maintenance Phase.

The principle criteria for model selection will vary throughout the life cycle. However, one criteria stands out for all phases, that being what information is available. A prototype architecture for a knowledge based criteria scheme supporting reliability model selection would include:

1. Basic Assumptions:
    a. Initial Fault Content (Known/Unknown).
    b. Fault Occurrence Independence (Yes/No).
    c. Fault Timing Independence (Yes/No).
    d. Fault Removal.
        - New Fault Introduction (Complete/Partial).
        - Removal Time (Negligible/Duration).
    e. Testing Environment.
        - Random.
        - Predetermined.

2. Formula Type (Linear/Logarithmic Poisson/Bayesian/etc.):
    - Curve shape (i.e. over estimate early).
    - Basic mathematical theory involved.

3. Parameter Estimation (for initial data requirements of model):
    - Which ones required by each formula (checklist).
    - Degree of confidence of each parameter if estimated.

4. Model Applicability:
    - For various life cycle phases.
    - For correct answer (i.e. time to test done, fault density, etc).
    - For Project Management answers (cost, schedule, and performance).

The actual creation of the knowledge base according to the above underpinnings is beyond the scope of this project. Obviously, the knowledge base cannot be of a simple monolithic design, but must be multidimensional. An object hierarchy for the reliability model selection rule base should be based upon the above criteria developed for each model. A knowledge base supporting the reliability model selection would be represented as a tree, with parameters at the bottom and the software reliability models at the top. The reason for this representation is to improve efficiency of the inferencing process by having a "fan-in" of knowledge (goal reduction). The prototype design is developed and presented in chapter 7.

For this knowledge representation, either forward chaining or backward chaining would be appropriate depending upon the question asked,. The inferencing would be forward if the parameters were known and a model must be selected. Conversely, search would be backward if the model was known based upon a desire to have a specific answer (i.e. testing time left), and it must be determined what reliability model must be chosen to given certain initial parameters. Additionally, there are times when so little information is known, that a combination of forward and backward chaining would be required to "find" acceptable compromises between parameters and models. This concept is further explored in chapter 7.

4.5 An Example

The reliability models most often referenced and discussed are the Musa Execution models. For this reason, this model will be used for some detailed discussions of the knowledge based selection of a software reliability model. Two models make up the Musa's execution model:

1. Basic Execution Model - linear
2. Logarithmic Poisson Execution Model - nonlinear

Knowledge base development usually follows one of two approaches. For seemingly unstructured domains (i.e. medical expert systems) where rules of thumb preside, an "ad hoc" rule base is developed. The advantage to this approach is how it handles the chaotic/random information. The disadvantage is that the rule base must be "tuned" since rules can often contradict each other in a "transparent" manner. The other approach is to organize the knowledge into a structured format (i.e. decision tables), or a least to arrange the knowledge as best as possible. The advantages of the more structured approach include efficiency, easier maintenance, and reliability. The biggest disadvantage is that the knowledge must be in a form that can be "formatted". Later, it will be seen that the Expert System chosen has a sophisticated built-in facility for knowledge base development and tuning.

Here, some of the information is available for the Musa Execution models that were chosen for this example. Again, for a complete KBQA, this information should be available for ALL the reliability models. For the execution models:

Parameter	Basic	Logarithmic Poisson
Initial Failure Intensity	$Lambda_0$	$Lambda_0$
Failure Intensity Change:		
Total Failures:	v_0	-
Failure Intensity Decay	-	Fee

For both models, values are needed for the initial failure intensity $Lambda_0$, and the failure intensity change. $Lambda_0$ is identical for both models, but the failure intensity change is different. The basic model requires the value for total failures. Initially, this can be **predicted** (before program execution) from characteristics of the program itself and of the software development process (elaborated in chapter 6). Later, it can then be **estimated** once a program has executed long enough so that statistically significant failure data is available (elaborated in chapter 5). For the logarithmic poisson model, the failure intensity decay parameter, Fee, is required. This also has predictable and estimatable procedures. But assuming that we are in the measurement phase, this information is known. Some example rules focusing on use of the Musa Execution model during later stages of the life cycle might look like this:

RULE 001: If a more accurate estimate is available for the Total Number Of Failures parameter, in comparison to the Failure Intensity Decay parameter, then there is evidence to support the use of the Basic Execution Model over the Logarithmic Poisson Model.

RULE 002: If a high predictive validity is needed early in the period of execution, and if the program is expected to be used with a decidedly nonuniform operational profile, then there is evidence to support the use of the Logarithmic Poisson Model over the Basic Execution Model.

RULE 003: If prior to program execution it is desired to conduct studies or make reliability predictions, then there is evidence to support the use of the Basic Execution Model over the Logarithmic Poisson Model.

RULE 004: If a value for Mean Time to Failure (MTTF) is desired, then there is evidence to support the use of the Logarithmic Poisson Model over the Basic Execution Model.

Of course, these rules are overly simplistic for illustrative purposes. In a realistic situation, it is expected that *some* predicted and estimated values for both of the parameters would be known, but to varying degrees of certainty. Thus the expert system would really have a certainty factor value (C.F. - degree of confidence) for each of the parameters. The example above was illustrated with an unrealistic binary decision. The KBQA would have to take several additional issues into account, one being the stage of the life cycle.

More rules can be based upon what we are expecting to get out of the use of the reliability model. This decision table is taken from Musa:

Purpose of application or existence of condition.	Basic	LP
1. Studies of predictions or existence of condition before failure data taken	X	
2. Studying effects of new s/w engineering technology (through study of faults; which must be related to failure intensity)	X	
3. Program size changing continually and substantially as failure data is taken	X	
4. Highly nonuniform operational profile		X
5. Early predictive validity important		X

The above preceding discussion was based upon the execution model. Still more rules can be based on the differences between the execution and calendar models based upon what you are looking for, and what information is available. The calendar model is based upon a "debugging process model". Information required for this model is:

1. Resources **used** in operating the program for a given execution time and processing an associated quantity of failures.
2. Resource quantities **available**.

3. The degree to which a resource can be **utilized** (due to bottlenecks) during the period in which it is limiting.

The benefits of using the calendar model is to develop solutions useful for planning schedules, estimating status and progress, and determining when to terminate testing. Parameter information required for this model are:

1. Planned - involve resource quantities available.
   - Ascertained from managers or project planners.
   - Determined by formula or experience.
2. Resource usage - involve resources quantities required.
   - Batch versus interactive debugging.
   - Debugging aids available.
   - Computer used.
   - Language used.
   - Administrative and documented overhead associated with corrections.

A significant level of detail is required just to develop a small example knowledge base for the Musa models, let alone development of a knowledge base for the other common models (as previously listed). Upon a detailed analysis of the various reliability models, it may be possible to eliminate some of them should similarities exist.

4.6 Knowledge Base Scope

Extending the scope of software quality to cover the complete spectrum versus just specializing on reliability, it is easy to visualize what a complete KBQA can deliver. Musa identifies some of the questions which can be answered:

System Engineering Applications

1. Specifying software quality to a designer.
2. Estimating cost of failure for an operational system.
3. Pricing a service.
4. Helping establish the market window for a software product (or product containing software).
5. Investigating trade-offs between software quality, resources, cost, and

schedule.

6. Selecting the failure intensity objective.
7. Determining the amount of system testing required.

Project Management Applications:

1. Progress monitoring.
2. Scheduling.
3. Investigation of alternatives.

Musa also defines five key roles in software quality engineering. *The expert system could be tailored to provide a different interface and consultation session based upon the individual needs of each role.* Specific needs of each of the Musa role are:

1. Managers

- Monitoring the status of projects that include software.
- Predicting the release date of software.
- Judging when to allow software changes in operational systems.
- Deciding on what software engineering technologies to apply to a project.
- Deciding whether to accept delivery of subcontracted software.

2. System Engineers:

- Automated/assisted software reliability model selection and tuning.
- Trade-off's between reliability, resources, and schedules.
- Investigation of options for allocation of reliability between components of the system.
- Deciding on what software engineering technologies to apply to a project.
- Impact of a proposed design change on reliability.

3. Quality Assurance Engineers:

- Automated data collection procedures that are needed to measure software reliability.
- Automated/assisted test data processing.
- Identify initial parameter values.

- Process resource expenditure and failure data.

4. Test Team:
   - Automated (or at least assisted) test plan generation.
   - Automated failure records keeping.

5. Debuggers:
   - Automated record keeping.
   - Automatic communique dispatching to others regarding fixes.
   - Data base of faults for future "tuning" of the software development process.

4.7 Summary

To develop a fully functioning KBQA, little research must be performed in the area of new software reliability models. There is a sufficient number of adequate models available to support the basic needs. To fully develop the KBQA concept for the *measurement* of reliability, the following are the very first tasks that are required to be performed:

1. Exhaustive analysis of the presently available software reliability models such that a structured decision table (or as structured as possible) representation can be developed. It is expect that some of the models can be eliminated due to redundancy.

2. For the decision tables, develop a rule base supporting the criteria required for the models to work versus the answers that are required of them.

3. Development of a control structure which allows for the optimization of model selection based upon the relative certainty of information available.

4. Development of rules to optimize the selection of reliability models based upon life cycle considerations.

CHAPTER FIVE

Reliability Estimation

CHAPTER FIVE - RELIABILITY ESTIMATION

5.1 Introduction

In the previous chapter, software reliability measurement models along with a complementary knowledge base selection approach was discussed. Expert System assistance in optimizing the software reliability model selection is a significant advancement, however, more can be done. The key to successfully integrating reliability engineering into the software development process is to calculate reliability even earlier in the life cycle. We must be able to estimate reliability immediately after coding has been initiated. In this chapter, reliability estimation will be addressed. Using advanced statistical trend analysis techniques, we are able to employ reliability estimation procedures. This involves accurate parameter estimation using "partial system" (partial execution/testing) data. As with software reliability measurement, Expert System assistance can be used to improve the estimation process. In the next chapter, the prediction of software reliability will be considered.

5.2 When to Estimate

Actually measuring the software reliability of a program in operational use is the most accurate determinate of reliability. With the goal of handling reliability as early as possible in the life cycle, reliability must be determined before the system is in the Operational and Maintenance phase. For our definition in this project, estimation can be performed only when some parts of the program are executable. Parts may be proper subprograms themselves, or may even be modules which are exercised with specially built programs called stubs and drivers.

This indicates that reliability assessment can occur as early as the Unit/Unit Testing, and continue into the Integration/Integration Testing and System Testing phases of the software development life cycle. Since without a finished product reliability measurement is not possible, we can settle for the next best thing - reliability estimation. In *this* project, reliability estimation is defined as the statistical inference procedure which is applied to failure data taken from partial program execution, to estimate software reliability.

A conceptual barrier may exist for those who have not been exposed to elementary statistics, or even doubt the use of such techniques for estimated purposes. However, sufficient evidence exists to support such techniques in all fields from medical, production line, to even software. Of course, the accuracy is directly proportional to the sample size and the selection techniques. There are two schools of thought which exist in sample selection. One concept emphasizes the randomness of the sample with the goal of insuring a representative coverage of the system. The other philosophy stresses concentration on "error prone" modules. This project does not recommend one selection approach over the other. It is recommended that the KBQA support both theories and keep a history of the eventual accuracy of both. It is a possibility that characteristics can be drawn which support either method based upon the prevailing circumstances.

In the Unit/Unit Testing and Integration/Integration Testing phases, partial program execution is presently performed. The normal practice for testing such partial program execution is by the use of suitable driver/stubs. However, the results of these tests are normally used for correcting bugs, versus being used in the reliability assessment process. This is a significant advantage of the KBQA approach.

In a way, the System Testing phase can be more difficult to determine the validity of results, even though the system can be tested as a whole. Should the operational environment be such that it is impossible to accurately emulate or simulate it, the reliability estimate will be of questionable quality. Of course, should the operation environment be completely simulatable in the System Phase, the result could actually represent the reliability measurement itself.

5.3 Module to Module Differences

The most significant obstacle to software reliability estimation is incorporation of failure severity propagation throughout different modules. The effects of a failure of a module after it has been incorporated into the final program may be varied. The failure of the individual module may result in overall system failure, or may just be localized and result in a failure which goes unnoticed. Thus, effects should be classified by severity of the failure. This issue can be conceptually expanded to address a failure in a simple text editor in comparison to a fuel cooling system for a nuclear power plant. It is both the effect that the system will experience if that module fails, and the effect that system failure will have to other things.

Essentially, two different failure classifications must be addressed here. The first is the overall impact of the program's failure upon things other than itself. Three classification criteria that are in common use are:

1. Economic Impact - expressed in terms of repair, recovery, lost business, and disruption.
2. Human Life Impact - effects due to different systems: nuclear power plants, air traffic control systems, military systems, etc.
3. Service Impact - i.e. interactive data processing system or telephone switching system.

The other failure classification is the effect of a module's failure on the operation of the program itself (i.e. system crash versus just a missing function). Significant research has gone into this area to determine a strategy for solving the propagation issue. The basic approach is:

1. The system should be divided into a set of components, each of whose reliabilities is known or is easy to estimate or measure.

2. Next, relationships between the reliabilities of components and the reliability of the system are addressed.

The relationships of modules to each other are analyzed and used to predict system reliabilities from module reliabilities. The analysis includes an allocation or a budgeting of a system's reliability to is components. Of all the techniques, the preferred method to handling severity classifications in estimating reliability and related quantities is proposed by Musa: "Classify the failures, but ignore severity in estimating the overall failure intensity. Develop failure intensities for each failure class by multiplying the overall failure intensity by the proportion of failures occurring in each class."

The KBQA must be able to address this issue to successfully estimate a system's reliability during the Unit/Unit Testing and Integration/Integration Testing phases. A knowledge base must be developed that will handle the module to module differences for both the above failure severity classifications. The easier classification to handle is the impact on economic, humans, or services by a failure. Development of the rule base to handle the effects of a failure on the system itself, and base the failure by severity, will be challenging.

5.4 System Configuration

Another obstacle to accessing reliability arises in very large systems where all the code is not exercised. It is also difficult during testing to have complete coverage of the code. As an example, some systems have to operate under more than one "operational mode". An example is a telephone switching system which may be operated in either a business or residential customer mode. In this case, the situation can be considered as having two separate "systems", since a significant portion of the system (i.e. modules and subprograms) is never used in one configuration or the other. The result is that latent failures may remain hidden, either to show up at a later date, or remain hidden forever.

Yet another difficulty can occur when attempting to estimate the reliability of such diverse systems as:

- Concurrent Systems versus
- Sequential Systems versus
- Distributed Systems versus
- *Standby Redundant Systems versus*
- Fault Tolerant Systems

Using conventional technologies, the complexity of such systems results in the actual measurement of the reliability of such systems being impossible. The KBQA using statistical estimation techniques and intelligent module selection (e.g. random versus failure prone) may have a chance of getting a handle on this problem. A more sophisticated methodology for the KBQA must be addressed here, else in these cases the reliability estimation may always be suspect.

As with the problems encountered in module to module interaction, the difficulties of assessing the accuracy of different system configurations are due to the combinatorial effects of "different" systems working in cooperation. This is yet another area where the KBQA must be expanded to.

5.5 Is Accurate Estimation Possible?

Possibly. This is an area where the quality of the Test Run Selection is most important in determining the resulting quality and efficiency of testing. Due to the complexity of an early estimation of reliability in large and complex software development efforts, the capability of knowledge based assistance to assist here is enormous. The primary benefit of the KBQA concept here is by development of a capability for automatic test plan generation.

The problem of module to module differences by a Knowledge Based software Quality Assistance (KBQA) would require the understanding of why some components are more fault-prone than others. The knowledge base designed to handled such a question would have to address such considerations as:

- The function implemented.
- The development methodology employed.
- The capability of the designers (i.e. quality of personnel).

Notice how the information we are using to estimated reliability differs from information that was used for reliability measurement. The primary concern with reliability measurement was selection of the appropriate reliability models to "calculate" reliability. In estimation, the process is not so exact and has a tendency to rely on "rules of thumb". In the next chapter about reliability prediction, the process is totally trusting of these "rules of thumbs".

The two ways in which test run selection can be approached are deterministic and random. In the deterministic case, each test case and its order in the test sequence is specified with certainty. The KBQA should be able to help chose the input partitions, and determining how many test cases should be used in selection from each partition. In random cases, more than one test case is possible at any point, and each test case has a probability of occurrence at that point associated with it. Here to, the KBQA should be able to help chose the test case inputs.

Even before this, the KBQA should be able to determine which form of choosing test run selection is specified. Recent studies by IBM have implicated the superiority of random testing based on operational profile by a factor of 30X over structural testing for nine large IBM products. It would even be useful for the KBQA to have such testing accuracy knowledge.

5.6 KBQA Estimation Inventory

The architecture of the KBQA will be such that it is extensible to account for the other quality factors at a later date. The methodology used by the KBQA to assist in *estimating* the reliability (and eventually quality) of software is:

- Estimating Before System Test:
  - Design errors caught by design inspections.
  - Coding errors caught by software inspections.
- Estimating During Test
  - Statistical measurement techniques.
- Estimating reliability from failure data in test, based on models and expected use.

5.7 Summary

To develop the estimation functionality of the KBQA, some work will have to be performed in determining how the statistical trend analysis techniques will be implemented. Estimating the reliability based upon partial program execution is straight forward. The more difficult concepts to develop are how to handle the combinatorial effects of the various reliability classifications. Another difficult task is handling the module to module differences. The use of random or failure prone based unit/component testing is a start in handling different system configurations.

CHAPTER SIX

Reliability Prediction

CHAPTER SIX- RELIABILITY PREDICTION

6.1 Introduction

The software quality factor reliability is normally considered to be a measurement-type of process. By measurement-type, it is implied that the reliability can directly be measured in the operational environment. In the previous chapter, it was shown that reliability can be estimated in the Unit/Unit Testing, Integration/Integration Testing, and System Testing phases of the life cycle. Often, this is much too late in the software development process, for reasons of efficiency and practicality, to correct a situation gone wrong. The most efficient time to calibrate and correct reliability is as early as possible in the software development life cycle. Practicality enters the equation even earlier in the feasibility and design phases. If present technology will not support the capability ambitions with the required reliability, either alternative solutions can be cultivated, or the project abandoned altogether.

6.2 Why Predict?

Earlier, in chapter 3, the Reliability Component Concept was proposed. The three components of software reliability engineering, in proper order, were prediction, estimation, and measurement. Of the three components, the prediction of the reliability of the software system from the software itself (code plus environment) and development process characteristics needs the most attention. Prediction also is the least developed of the components, yet offers the greatest potential for success in terms of ultimate potential payoff. In addition, high quality early decisions have the greatest impact on project management factors (costs, schedules and performance issues).

6.3 What to Predict?

The prediction component is a composite of two key entities. One entity involves attempting to predict the input parameters and the other data requirements required to administer the reliability models. The use of the reliability models as pure prediction (in our terminology) may seem out of place since these growth models are normally used in measurement and to prediction/estimate (in the conventional use of the terms) the future based upon the past. However, the reliability models do provide some important information (i.e. Mean Time To Failures) that can be studied to see if the proposed development method and schedules will fit in with everything else.

The other aspect of prediction is to predict the reliability on the basis of characteristics of the software development process, from characteristics of the software itself, and from historical success with such products. The utility of using the software reliability models for pure prediction are debatable and need further clarification. However, use of the these yardsticks for the aspect of reliability prediction holds the most promise. The use of knowledge-based and expert system technology can be an integral part in the prediction process. The Knowledge Based software Quality Assistant (KBQA) can assist the prediction process in a analogous manner as was seen with estimation and measurement process.

6.3.1 Reliability Models:

The KBQA must be able to assist in the prediction of the parameters of all the software reliability models. The manner in which this could be implemented will draw heavily on the software reliability model categorization already performed for the measurement chapter. The key additions here are to focus the knowledge base to address the parameters themselves versus focusing on optimizing the selection of reliability models. After a detailed list of all the parameters from all the models is performed, the list should be categorized into groups based upon the confidence level assigned to prediction of that particular parameter. It can be assured that some parameters are more difficult and capricious than others, and are affected by other circumstances (i.e. life cycle). It is assumed that much of the information developed to recommend one reliability model over another (chapter 5) can be reused or slightly modified to assist in this area. If the knowledge base can support objected oriented constructs (e.g. Nexpert Object), then inheritance can be employed to greatly assist in reducing any duplication of effort.

6.3.2 Software Development Process:

The best understood methodology to assess the software development process of an organization is by use of the Software Engineering Institute's Capability Maturity Measurement (SEI/CMM) system. This system is available in text or electronic format (the questionnaire is available in an automated system which runs on IBM PC's and Apple MacIntosh's). Two approaches could be used to insert this capability into the KBQA. First, it could be coded from scratch - a rather tedious or straight-forward process. Conversely, if the KBQA architecture allowed "external" programs to run, the prior developed versions can be used. In the next chapter, it will be shown that the software and hardware suit recommended for the KBQA will allow incorporation of virtually any predeveloped software.

6.3.3 Software Characteristics:

Another manner in which to predict the reliability of the software is from the characteristics of the software product itself. There are several source code analyzers commercially available (i.e. AdaMat) which produces statistical analysis based upon factors such as size, complexity, modularity, general design (i.e. overuse of Global Variables), etc. These systems can be incorporated as is into the KBQA architecture as defined in the following chapter. Another measure can be based upon the application's domain analysis. Some domains put more stress on a system than others (i.e. uneven loading and "emergency" situations). A simple example is the stress placed upon the simple Management Information System in an office versus the Electronic Warfare system in a USAF Fighter in combat. This implies that the hardware execution environment must also be taken into account. Another factor involves predetermined software characteristics (i.e. Ada versus Assembler). There is a wealth of information available comparing various languages with fault density. Yet to be determined is the effects of design (i.e. Object Oriented versus Structured versus XYZ) on reliability. All these determinants must be accumulated with the KBQA's knowledge base.

6.3.4 Historical Factors:

Another aspect is whether or not the developers have been successful in similar ventures. This should encompass both the application domain (i.e. MIS versus scientific) and other products in general. It might be feasible for the KBQA to have a company/developer data base and keep records.

6.4 Example

As an example, it will be assumed that the system is presently in the stages of the life cycle prior to code execution. This implies that we can be anywhere from the Feasibility phase up to and including the Unit (coding), but not the Unit Testing phase. In a consultation session, the KBQA must address the following issues in predicting the level of reliability before program execution:

1. From characteristics of the software product.
2. From the software development process.
3. Historical Factors.

In addition to the above three, assume that some additional information that is generated by the software reliability models is also desirable. This example will taxe the full capability of the KBQA.

First, assume that the KBQA recommended the Musa Basic Execution Model based upon a particular request for information during the consultation session. When the program is not yet executable, the execution time component parameters must be predicted from program characteristics rather than estimating them from failure data. For the basic execution time model, the parameters that are required are:

- The number of inherent faults w_0.
- The fault reduction factor B.
- The fault exposure ratio K.
- The linear execution frequency f.

The KBQA could predict the number of inherent faults in a number of ways. Two of the more common are based upon size and complexity metrics. There are a number of studies that have found a relationship (but somewhat weak in that there are wide variations) between the number of faults and code size. In defense of the metrics, a semi-intelligent guess is better than no guess at all. Other metrics are based upon the complexity of the program. Where size metrics are limited in imaginative deviations, complexity metrics have many different and novel approaches to measuring complexity.

Predicting the number of inherent faults should not be limited to analysis of the software product itself. *A more accurate measure of the number of inherent faults can be determined by the software development process* - the kind of process that Watts Humphrey at the SEI and Herb Krasner at SAIC are working. The most representative evidence of this is in a comparison of the Japanese and American automotive production lines (and typical of other products also). As a group, the typical Japanese automotive company in comparison with a typical American company exhibits the following:

1. Produces a vehicle with higher quality.
2. Employees far fewer quality assurance personnel.

The ever quality conscience Japanese do not seem to employee many quality assurance personnel at the *end* of the production line. Yet the cars leave the plant with fewer J.D.

Power measured (Initial Quality Survey) reports of defects! If they are not checking for quality at the end of the production line as thoroughly as the Americans are, where is the quality coming from? There is overwhelming evidence to believe that the quality is acquired during the **process**, NOT after.

A strong approach for allowing the KBQA to predict the number of inherent faults would be based upon a unification of the number of faults predicted from the above means. A knowledge based approach would be able to determine which factors should be weighted heavier than the other. Through a minimal consultation session (possibility eliminated by automatic code/scenario analysis), the prediction should be accurate and straight-forward.

The next factor to consider in the Musa Basic Execution Model is the fault reduction factor B. Based upon current studies, Musa believes that the initial measurements of the fault reduction factor show a strong tendency to be relatively stable across different projects. There are no other references concerning this reduction factor other than data collection surveys of the raw numbers them selves. It may be prudent to tie this number in with software process analysis as discussed above. The dangers of using a stable fault reduction factor will be over/under estimation based upon the individual process of the software developers.

Obviously, the capability of achieving a high fault reduction ratio is based upon the "software repair and debugging process". Then again, the capability of reducing faults is proportional to the quality of the software design. For this parameter, the SEI CMM levels could predict the levels to be achieved.

The third factor that the KBQA must determine for this software reliability model is the fault exposure ratio K. It may be possible to tie this factor in with the fault reduction factor. The success of both of these factors appear to be a by-product of the quality/quantity of the V&V/Testing process. Musa has stated that the fault exposure ratio K shows some dependency on the structure of the program, and the degree to which faults are data dependent. As with the fault reduction factor, the fault exposure ratio may average out for large programs.

The linear execution frequency f, is easily determined by division of the average instruction rate by the number of object instructions in the program. Knowledge of the values of these parameters are important if we are to obtain reasonably accurate predictions of software reliability. By looking at this one specific reliability model, it should be apparent that reliability prediction should be heavily based upon the software development process.

Other factors exist beyond the parameter prediction for the reliability models. Software reuse will also have a strong influence. The issue here is complicated since just the raw percentage of software reuse is insufficient. A better measure than raw percentage of reuse may be subsystem, segment, or even component reuse percentages. Just predicting the reliability effect of "high quality" reused software does, by itself, not take into account the intra-module effects versus inter-module combinatorial factors of reused software statements/parts versus complete components.

6.5 Static Analysis

Another method of predicting reliability prior to program execution (or unit execution) is derived from Code Inspections or any form of static code analysis. The KBQA could be able to log and classify defects found by code inspectors. Automation of this process could allow a relationship to be formed based upon the kinds and number of defects, and how such effects reliability.

6.6 Summary

Software reliability prediction as proposed by this project is more of an art than a science. The biggest challenge to the KBQA will be in the prediction area (versus the estimation and measurement areas). Several other issues can be "thrown" into this territory. For example, the reliability of the function implemented can be statistically determined from two angles. First, the analysis of similar functions by domain (i.e. mathematical versus logical versus etc.) can offer a rough estimate of reliability. The knowledge base should be designed such that it collects historical experience in the data base, along with a mechanism of continual "fine tuning" of itself. Lastly, the knowledge base must be able to assign confidence weights to each of the above mentioned constituents and arrive at an overall reliability estimate. Another thought is whether Fuzzy Logic mathematics can assist the knowledge base to arrive at conclusive results given such indeterminate input data.

CHAPTER SEVEN

Knowledge Base Architecture

CHAPTER SEVEN - KNOWLEDGE BASE ARCHITECTURE

7.1 Introduction

The key to successfully integrating reliability engineering into the software development process is to identify how we can determine reliability early on in the life cycle (versus first specifying reliability, then measuring it). In this chapter, the architecture for a knowledge based expert system to support the reliability-engineering process will be explored. Even though this project concentrates on only the reliability aspect of software quality, the architecture of the Knowledge Based software Quality Assistant (KBQA) will be such that it is extensible to account for the other software quality factors at a later date. The methodology used by the KBQA to improve reliability-engineering of software is:

1. <u>Predicting</u> Reliability Before Program Execution:
    - From characteristics of the software product such as:
        - Domain analysis.
        - Hardware execution environment.
        - Predetermined software characteristics (i.e. Ada versus Assembler).
    - From the software development process.
        - Software Engineering Institute's Capability Maturity Measurement.
        - The training/level of expertise of the developers.
    - Historical Factors.
        - Developers performance on prior efforts in general.
        - Developers performance on prior efforts in this domain.
        - State of the art in this domain (developer independent).

2a. <u>Estimating</u> Before System Test:
- Design errors caught by design inspections.
- Coding errors caught by software inspections.
- Any other documentation errors.

2b. Estimating During Test:
- Statistical trend analysis techniques based upon partial measurement.

3. <u>Measurement</u> After Deployment:
    - Actual reliability in the operational environment.

- Feedback of "lessons learned" back into the software development process.

7.2 System Architecture

The driving force behind the proposed system architecture is two fold. First is system availability. The computer systems that are readily available from both an economical and practical perspective are the MS-DOS/Windows-based Personal Computer (PC) and the Apple MacIntosh. Capable system configurations are available for under $5000. Due to the widespread use of both machines, purchases would probably not be necessary since the systems are likely to be available at most locations already. The second goal is to provide as much software portability as possible, while not hindering the technical sophistication of the system. There are various advanced Commercial-of-The-Shelf (COTS) software packages that are not only available for the PC and the MacIntosh, but has also been ported to most other computer systems as well. Any software that must be written to support the system will be sparse, yet if written correctly will also yield results that are also portable.

In summary, the goals of the system architecture are low cost and portability. The recommended system configuration, along with explanations for the KBQA are as follows:

HARDWARE:

Apple MacIntosh (preferred).

- Minimum of 4 MB RAM.
- 40 MB Hard Drive minimum.
- Color capable.

or

IBM PC (386/486)

- Minimum of 4 MB RAM.
- 40 MB Hard Drive Minimum.
- Color capable.
- Mouse.
- MS-Windows.

SOFTWARE:

Graphical User Interface: HyperCard

- For all MMI (other software will be transparent to the user).
- Best package available for ease of construction and simplicity of use.
- Available only on the MacIntosh, but several "clones" are available for other systems (minimal porting effort).

Expert System Shell: Nexpert Object

- Available on most architectures.
- Robust system (object oriented, forward/backward chaining, extensive interface routines, etc).
- Support a wide variety of interfaces; Ada/C++, Hypercard, Oracle database, etc.

Data Base Management System: Oracle

- Support of the requirements-specific traceability of reliability.
- Support of the requirements-specific monitoring of reliability.
- SQL-based for portability.
- Available for most systems.

Programming Language: Ada (C/C++ alternatively)

- Most of the software reliability models are already available in source code format (i.e. Musa) at minimal cost.
- Very few of the models will have to be coded.
- Alternatively: It may be preferred to recode all the reliability models so that the various models can be indexed/grouped by model type.

7.3 Knowledge Base Architecture

The knowledge base should be developed in a modular approach. Instead of having a single knowledge base containing all the rules, it is recommended that the KBQA employ various knowledge bases which will be specialized for specific purposes. Sectioning the knowledge base into several focused knowledge bases allows for a more modular approach to knowledge construction, along with simplified maintenance. Other advantages of a modular knowledge base construction are:

- Specialization of various types of knowledge representation:
    - Functional-Based Reasoning.
    - Case-Based Reasoning.
    - Model-Based Reasoning.
- Multiperson construction of the knowledge bases:
    - Experts can concentrate on developing knowledge only in their respective areas of expertise.
    - Allows several knowledge base developers to work concurrently.
- Increased performance during consultation sessions:
    - Directed inferencing improves efficiency.
    - Finer grain control over the reasoning process.
- Easier Testing/Validation & Verification of the individual knowledge bases.
- Support of a "software engineering" approach to knowledge base construction.

There are presently three modular architectures under consideration for the KBQA. This project will make no attempt to recommend one approach over the other. It may be possible to converge on a consensus for one approach over the other once further study has been performed on the knowledge base development. The knowledge base insight required to justify such a decision does not exist at the present. The three approaches are:

1. Distribution of the knowledge into the different phases of the life cycle.
    - Pseudo-model based reasoning.
2. Distribution of the knowledge into the different "components" of reliability.
    - Pseudo-case based reasoning. Reference chapter 3 section 4.
3. Distribution of the knowledge into allocated tasks.
    - Pseudo-function based reasoning.

7.3.1 Architecture 1: Life Cycle

The approach treating the distribution of the knowledge into the different phases of the life cycle has a foundation based upon the significant differences in handling reliability in each phase. In the early phases of software reliability engineering, specifically the feasibility study, the requirements analysis, the preliminary design, and the detailed design, the actual measurement of reliability is impossible. However, prediction of reliability is certainly not out of our reach and is capable of being performed. Conversely, in the Operational and

Maintenance phase of reliability, prediction is impractical since total measurement is possible. It is obvious that reliability is handled differently in each phase of the life cycle. The creation of different knowledge bases to support the various life cycles can easily be done. The focus of each knowledge base at each phase would be comparable to:

- Feasibility phase:
    - Technological knowledge.
    - Domain knowledge.

- Requirements phase:
    - Domain knowledge.
    - See IEEE Special Issue on Requirements.

- Design phase:
    - Process knowledge.
    - Architecture knowledge.
    - Hardware knowledge.
    - Software knowledge.

- Coding/Unit Testing phase:
    - Knowledge on unit testing.
    - Knowledge about determining Reliability as function of Prediction/Estimation.

- Integration/Integration Testing:
    - Knowledge on integration testing.
    - Knowledge about determining Reliability as function of Prediction/Estimation.

- System Testing:
    - Knowledge on system testing.
    - Knowledge on operational emulation.
    - Knowledge about determining Reliability as function of Estimation/Measurement.

- Operational and Maintenance:
    - Knowledge on operational testing.
    - Knowledge about keeping reliability history/monitoring.
    - Knowledge about un-reliability limits/warnings.

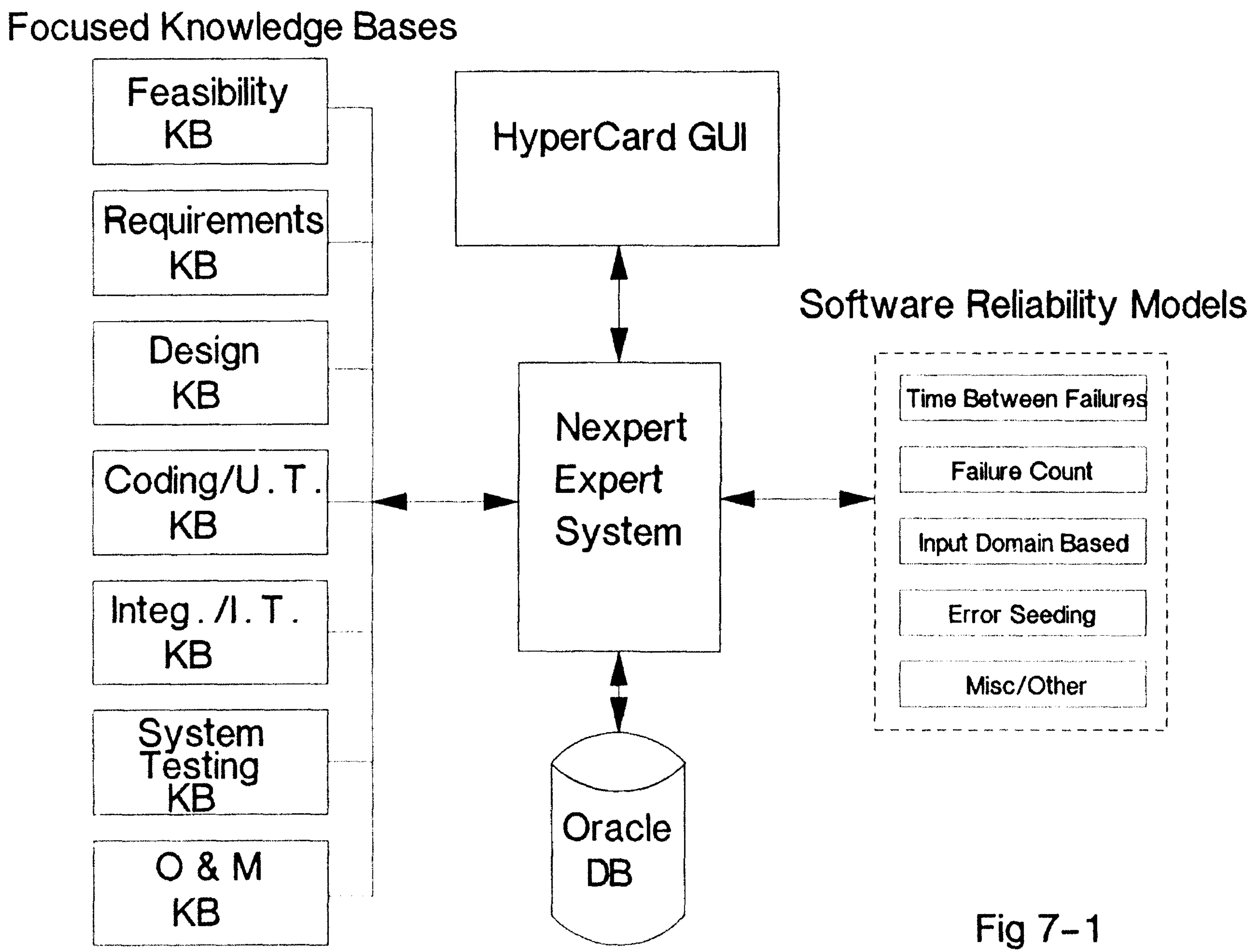

Focused Knowledge Bases
Feasibility KB
Requirements KB
Design KB
Coding/U . T . KB
Integ . /I . T . KB
System Testing KB
O & M KB
HyperCard GUI
Nexpert Expert System
Oracle DB
Software Reliability Models
Time Between Failures
Failure Count
Input Domain Based
Error Seeding
Misc/Other

Fig 7-1

The architecture for this view is shown in figure 7-1. In this setup, as with the others, Nexpert Object is the drivers seat for control. The knowledge of software reliability engineering has been placed into seven different knowledge islands, each one specialized for a different phase of the life cycle. For example, in the Operational and Maintenance knowledge base, we would not expect to find rules supporting the prediction of reliability of a model based upon size and complexity metrics. But it would be expected to find rules which optimized the selection of software reliability models for deployed systems. All user interaction is performed through the HyperCard Graphical User Interface (GUI). The Oracle Data Base Management System (DBMS) is primarily used to hold such information as:

- Requirements-specific traceability of reliability.
- Requirements-specific monitoring of reliability.
- Defect reports/Code modifications/etc.
- Project Management information.

The Software Reliability Models module containing all the models has an interface allowing the rules to determine the most appropriate model to use.

7.3.2 Architecture 2: Components of Reliability

Another approach in handling the distribution of knowledge is by focusing on the various "components" of reliability. In quantifying reliability, we can define it to be the weighted sum of various percentages of Prediction, Estimation, and Measurement. This sum must equal one. For example, during the Feasibility phase, the only measure of reliability available is that of prediction. So reliability in that case could be specified as 100% Prediction. In a different phase, coding and unit test, reliability is based mostly on estimation and prediction; say 25% prediction and 75% estimation (these numbers are not meant to define reliability at this phase, it varies from case to case, consider this just to be representative of an imaginary program). Lastly, we would also require a domain knowledge base since it may be beneficial to separate this into a specialty also. The focus of each knowledge base at each phase would be comparable to:

- Domain:
    - Application domain knowledge.
    - Considers recommended reliability (specification).
    - Considers the hardware/software domains.
        - e.g. Ada versus FORTRAN.
        - e.g. Space-based versus naval-based versus ground-based.
        - e.g. SEI Capability Measure Model rankings of 0 to 5.
        - etc.
    - Physically the largest knowledge base.

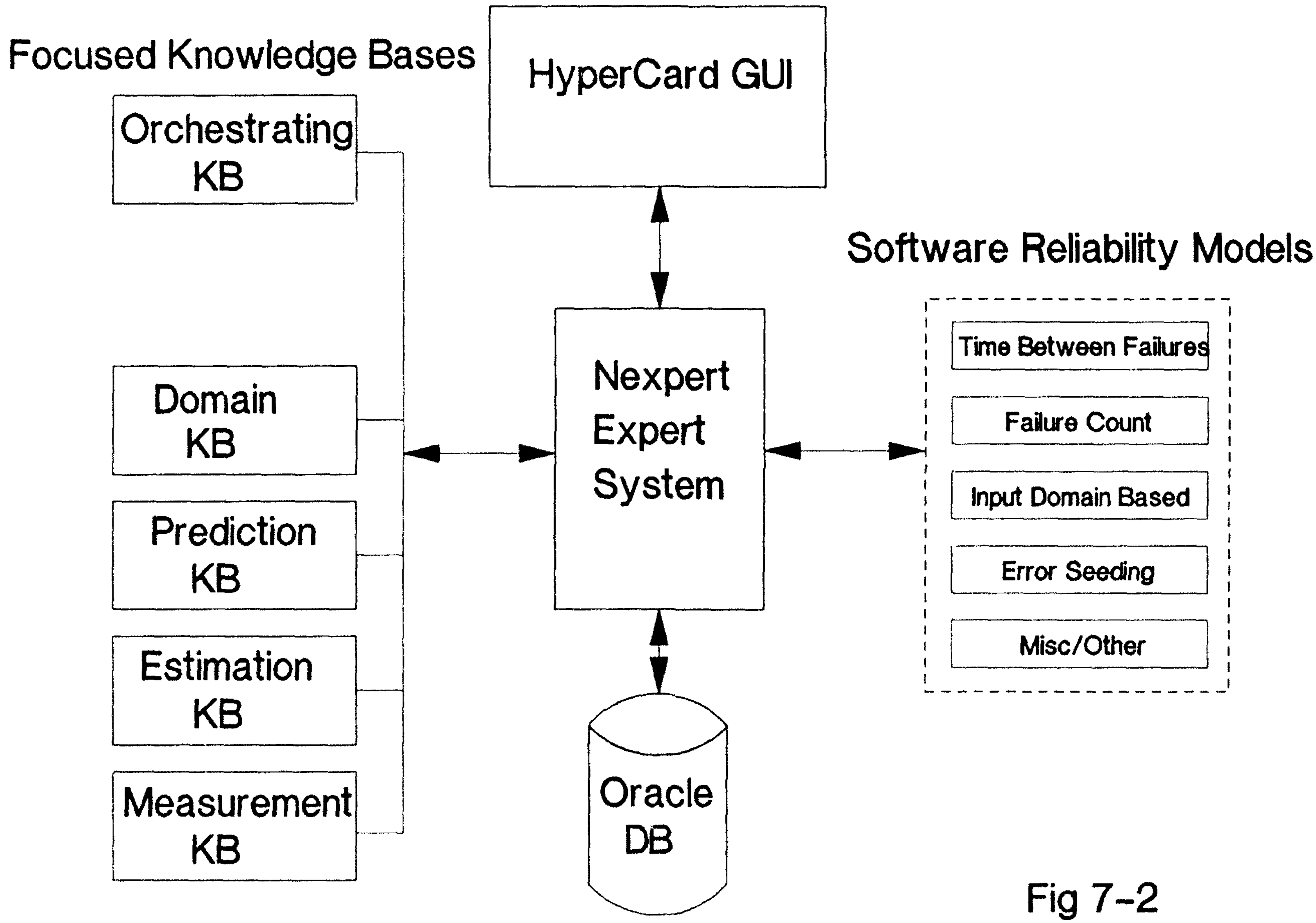
- Prediction Component of Reliability:
    - Considers what stage of the life cycle.
    - Extensive reliance on the Domain knowledge base.

- Estimation Component of Reliability:
    - Considers what stage of the life cycle.
    - Weak reliance on the Domain knowledge base.
    - Moderate use of the Reliability Models to estimate reliability.
    - Most control-extensive of the three knowledge bases.
        - Must interact with the other three knowledge bases.
        - Must balance the percentage of weighing between prediction and measurement.

- Measurement Component of Reliability:
    - Considers what stage of the life cycle.
    - No reliance on the Domain knowledge base.
    - Extensive use of the Reliability Models for measurement.

- Optional: Orchestrating
    - The control logic to allocate the balance between prediction, estimation, and measurement may become quite complex. This responsibility may be better suited in a knowledge base of its own and removed from the Estimation Component knowledge base.

This setup is shown in figure 7-2. Notice that the architecture is the same as the previous one except for the different specialization of the knowledge bases.



Focused Knowledge Bases
Orchestrating KB
Domain KB
Prediction KB
Estimation KB
Measurement KB
HyperCard GUI
Nexpert Expert System
Oracle DB
Software Reliability Models
Time Between Failures
Failure Count
Input Domain Based
Error Seeding
Misc/Other

Fig 7-2

7.3.3 Architecture 3: Development Tasks

Another approach is to manage the distribution of knowledge into allocated tasks or aspects of determining reliability. Besides the application Domain knowledge base, there would be a knowledge base for the Test Type, and another for Life Cycle considerations. The focus of each knowledge base at each phase would be comparable to:

- Domain:
    - Application domain knowledge.
    - Considers recommended reliability (specification).
    - Considers the hardware/software domains.
        - e.g. Ada versus FORTRAN.
        - e.g. Space-based versus naval-based versus ground-based.
        - e.g. SEI Capability Measure Model rankings of 0 to 5.
        - etc.
    - Physically the largest knowledge base.

- Test Type:
    - Determining the best test based upon such circumstances as:
        - Resources available.
        - Unit versus module versus integration.
    - Recommendations and explanations of specific test types.
    - Automatic generation of test plan.

- Life Cycle:
    - Determining the allocations given to:
        - Prediction.
        - Estimation.
        - Measurement.
    - Software Engineering process considerations.
    - Also would act as the "driver" knowledge base.

This setup is shown in figure 7-3. Again, the architecture is the same with the exception of the knowledge bases.

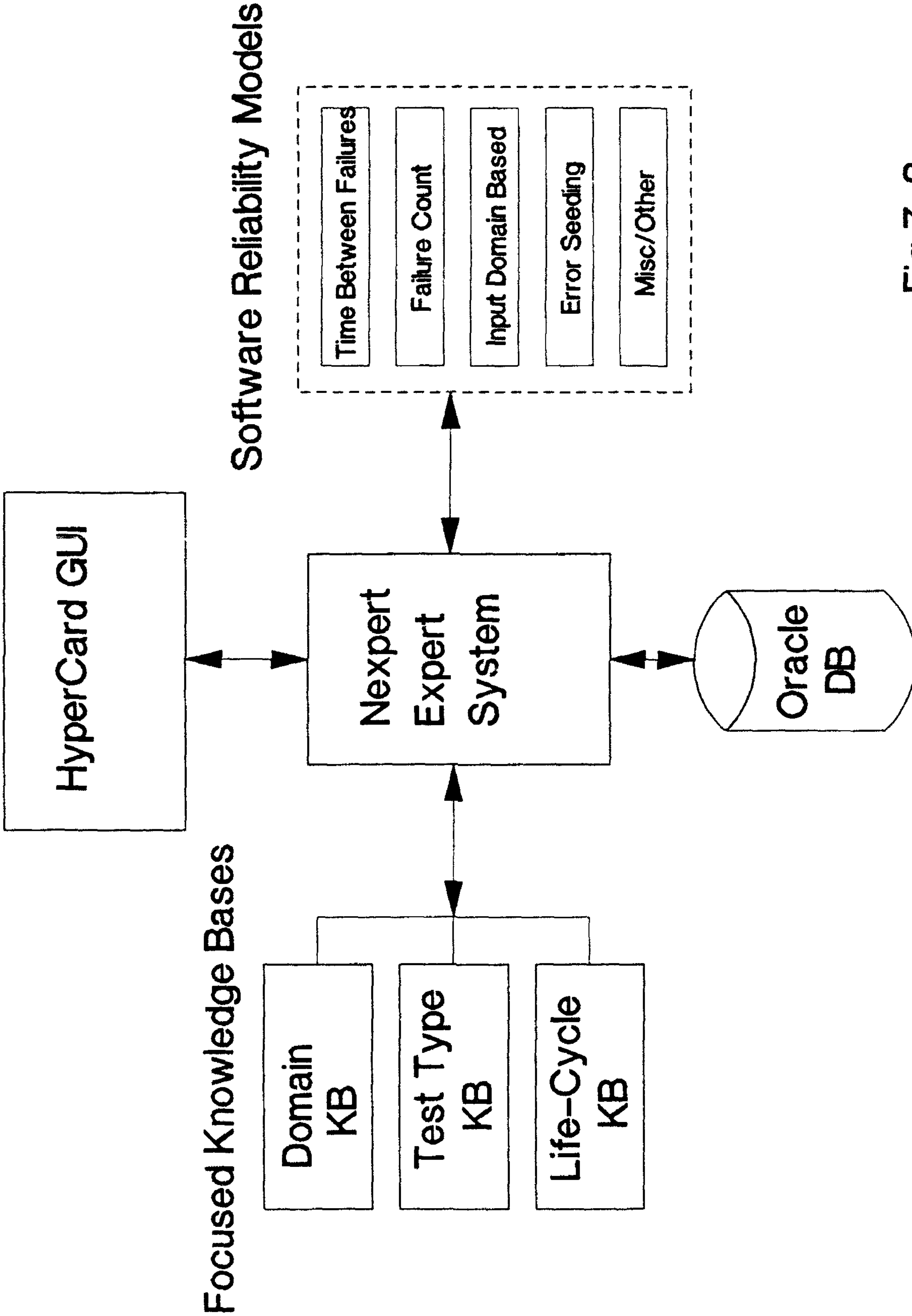

Software Reliability Models
Time Between Failures
Failure Count
Input Domain Based
Error Seeding
Misc/Other
HyperCard GUI
Nexpert Expert System
Oracle DB
Focused Knowledge Bases
Domain KB
Test Type KB
Life-Cycle KB

Fig 7–3

7.4 Advantages and Disadvantages

As mentioned earlier, a decision of one architecture over another will not be given in this project, much deeper analysis is required to do so. However, some high level arguments for and against each architecture can be performed.

Architecture 1 with knowledge partitioned by the different phases of the life cycle allows for modelling the knowledge based around the tasks involved in each phase of the life cycle. Advantages are primarily due to the close matching to DOD-STD-2167A and other project management scheduling issues. Disadvantages are the large amount of duplication of effort of the reliability components in each phase of the life cycle. It is yet to be determined if the duplication of effort is worth the ease of construction and close matching to project management.

Architecture 2 with knowledge partitioned by the different components of reliability is conceptually simple. Advantages include the minimization of the duplication of the components by life cycle. This is conceivably the most concise method of knowledge construction. However, the difficulty may be in the complexity and abstract nature of the knowledge base partitions.

Architecture 3 with the knowledge partitioned by the allocated tasks may match the activities of present software development better than the first two. Many organizations have software development procedures in place with activities scheduled in sequences. Disadvantages are primarily around the handling of software development as a series of predefined sequences. In addition, newer software process modelling techniques will probably not match the older ways of doing software development.

7.5 Knowledge Representation

The representation of knowledge is the fundamental binding between knowledge and the data collection which supports the reliability engineering process in the KBQA. This project assumes that a sophisticated knowledge representation scheme similar to the one in Nexpert Object is used to develop the knowledge bases. In figure 7-4 are the principal graphical representations use to support the construction of the knowledge bases.

A Class (i.e. a type) is represented by a circle, an Object (instance of a class) is represented by triangle, a Slot (a value) is represented square, and a Property (type for the slot's value) is represented by a rectangle. For further information on knowledge representation, see the Nexpert Object Functional Description documentation. This is an excellent tutorial on knowledge representation that will hold true for several of the more advanced expert system shells.

One of the first knowledge representation schemes that must be developed is for the various test types. Tests are generally divided between the structural class and the functional class. However, some testing text authors have defined a hybrid class which is a combination of the functional and structural classes. In this project, the nomenclature for testing will be derived from the text Software Testing Techniques, 2nd Edition, by Boris Beizer. For a detailed description (i.e. tutorial) of software testing, this book should be reviewed.

In figure 7-5, a test type hierarchy has been defined. All tests are defined to be sub-classes of the type "Test Type", and would inherit all the properties of that type. The structural testing class is further broken down into various structural tests. The dotted lines/figures represent more additional items. A complete description of the most common structural tests is provided in Beizer in section 3.3.7 along with the relative strength of each kind of test. The following knowledge would be expected to be instilled into the knowledge representation scheme:

- Various kinds of each structural test.
- The relative strength of each structural test.
- The amount of time required for each structural test.
- When to choose one test over another.

NEXPERT OBJECT REPRESENTATIONAL STRUCTURES

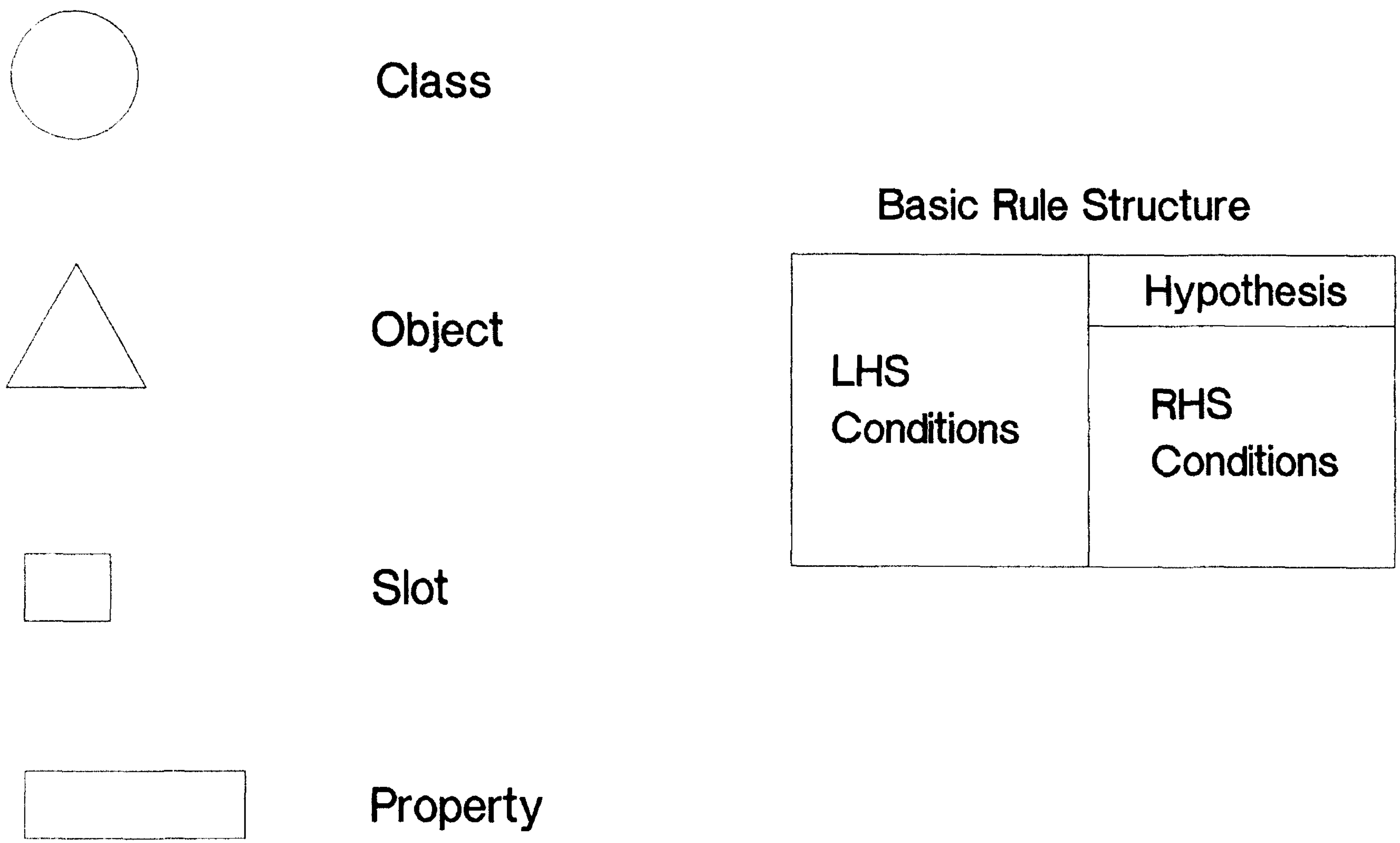


Fig 7–4

TEST TYPE HIERARCHY

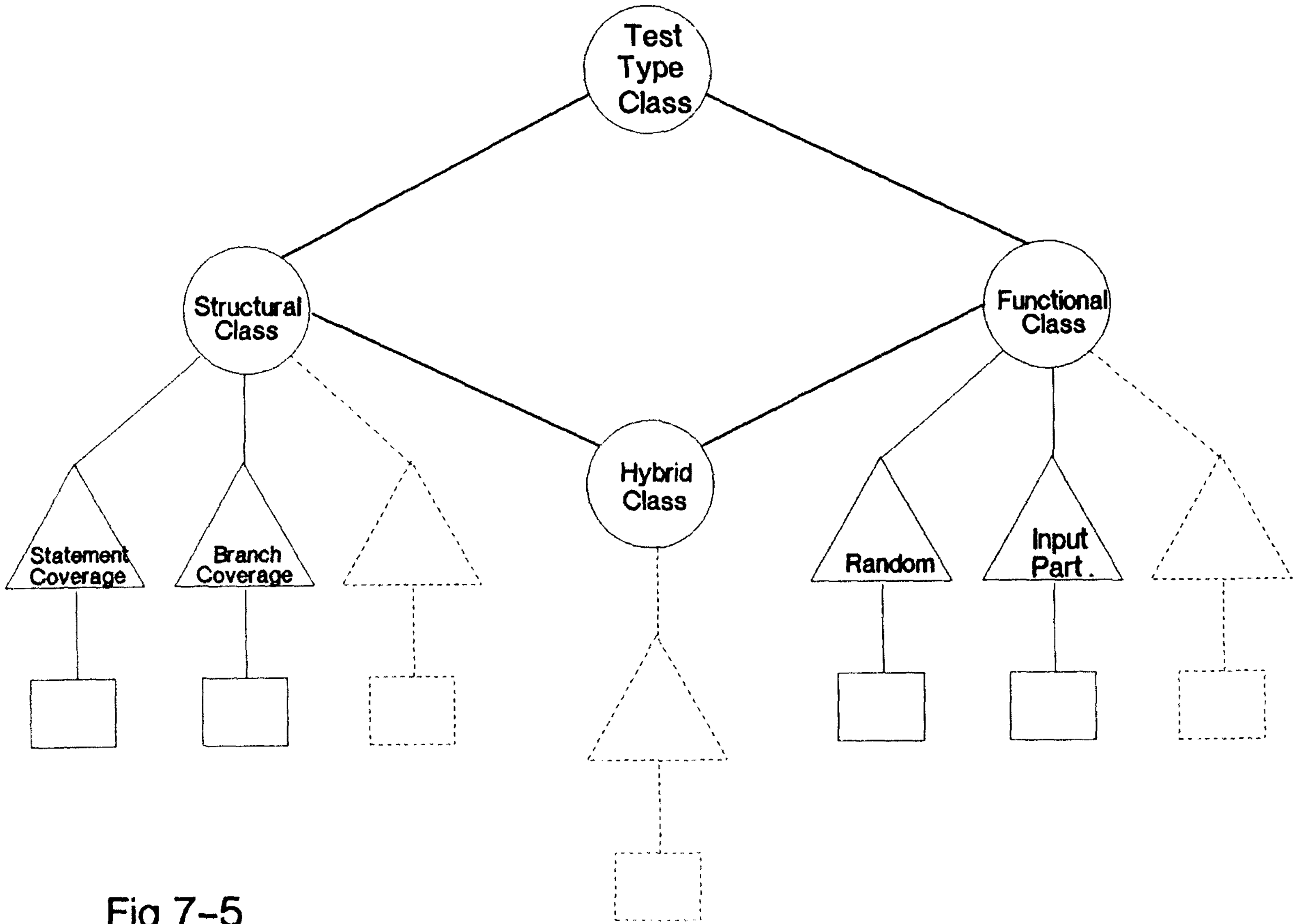


Fig 7-5

A similar knowledge representation in the KBQA should exist for the functional and hybrid classes of tests. Notice that the hybrid class development may be simplified by inheriting properties from the functional and structural classes. The KBQA should be able to determine what test types should be employed based upon such factors as:

- The desired reliability.
- The amount of resources (i.e. personnel, money, time) available.
- The module's function (i.e. logical versus mathematical) type.
- The stage of the life cycle.

Another hierarchy that must be developed will be used to support the various reliability models. This hierarchy is shown in figure 7-6. As discussed in chapter 4, the various reliability models can be sub-grouped in four class: Time Between Failures (TBM) models, Failure Count (FC) models, Fault Seeding (FS) models, and Input Domain Based (DB) models. This particular diagram has been expanded to show the Musa Execution classes. The P1 and P2 slots are used to hold the parameters that are required for the application of each model.

The KBQA must make decisions of the applicability of each model based upon circumstances such as function type, stage of the life cycle, estimation or measurement as the goal, etc. Some of the capabilities that KBQA must support are:

- Given a required result (i.e. testing time to failures), pick the model(s) which best support the answer.
- Given a list of input parameters that are available, pick the model(s) which can be used.
- Given a combination of the above two (i.e. answers desired and parameters available), provide a suitable reliability measurement collection (one or more models).
- Provide a prioritized list of the confidence of the application of each model given any combination of the above three.

RELIABILITY MODEL HIERARCHY

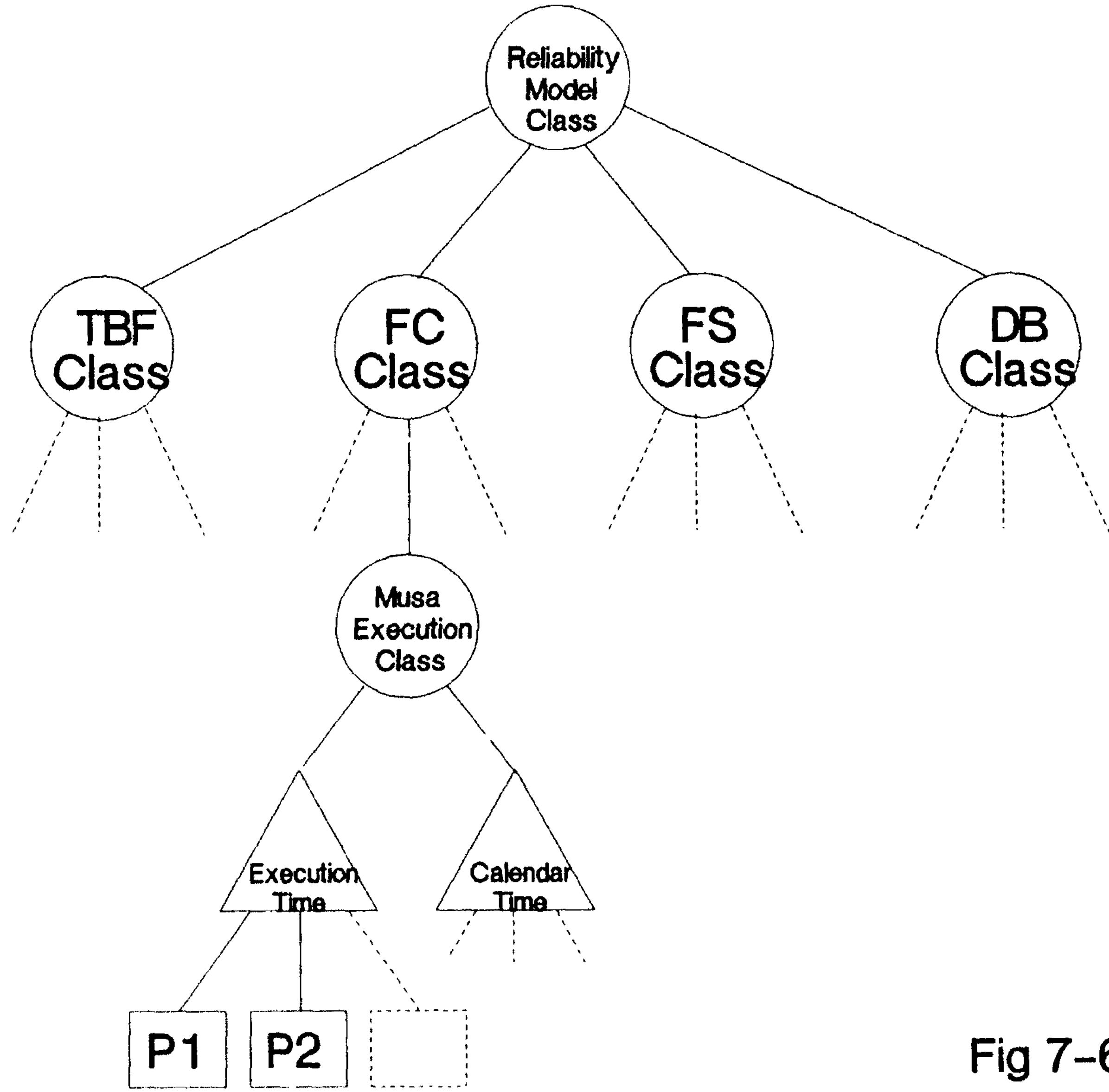


Fig 7-6

CODE/SYSTEM TYPE HIERARCHY

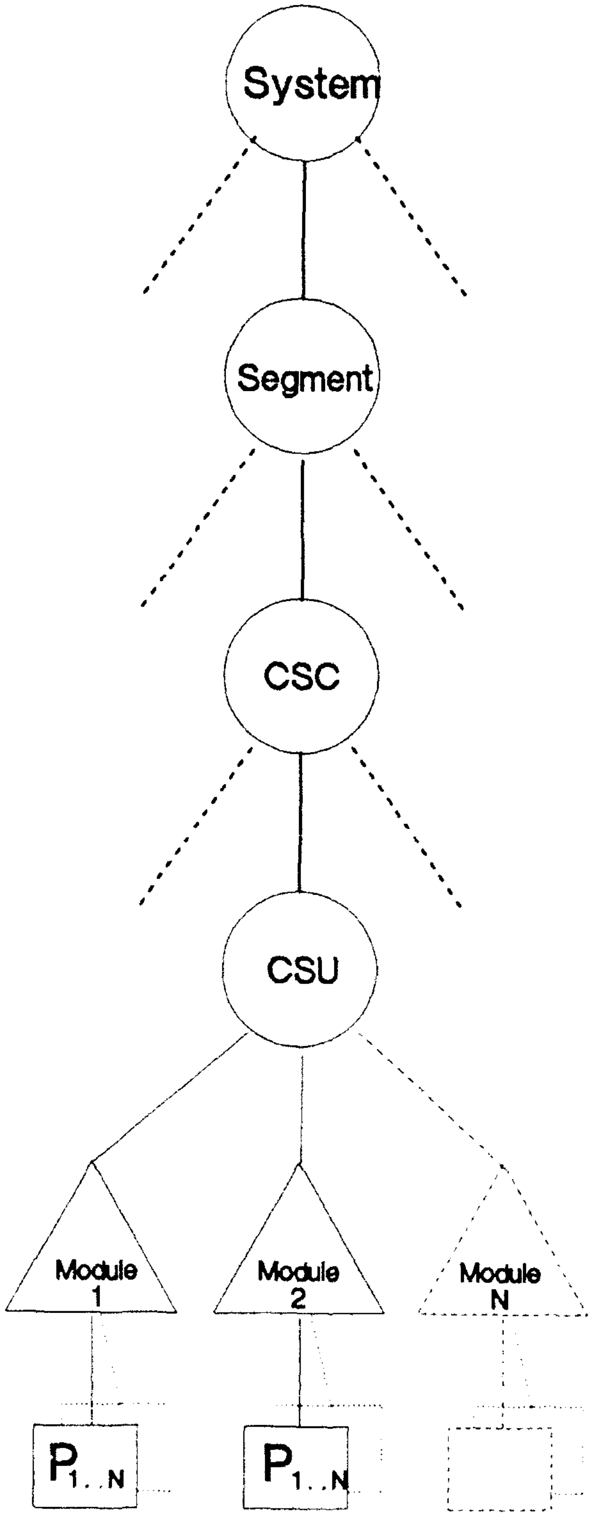


Fig 7-7

Another important hierarchy for knowledge representation is the division of code within a system. Figure 7-7 show a representation that is consistent with our earlier definition in chapter 3. It is the DOD-STD-2167A definition for a system, except for the proposed elimination of the CSCI division. The primary purpose of this architecture is to provide knowledge base information on the code's history, life cycle phase, bug reports, tracking information, traceability information, monitoring information, flow-graph, documentation, specifications, author(s), etc.

A key feature of all these hierarchy are their extensibility. Should it be desired to add additional or new test types, reliability models, etc., to the hierarch, the maintenance of the knowledge base is relatively straight-forward.

CHAPTER EIGHT

Automatic Test Plan Generation

CHAPTER 8 - AUTOMATIC TEST PLAN GENERATION

"Program testing can be used to show the presence of bugs, but never their absence!". Dijkstra

8.1 Introduction

Earlier, it was contested that the key to quality is to address the issue earlier in the life cycle. Even with increased attention to "building" in the quality, post coding testing is still necessary. The automatic generation of a test plan must address two issues. First, **what** tests are necessary to perform the testing. Second, **how much** testing must be performed. In the Knowledge Based software Quality Assistant (KBQA), we have the capability to extend test planning beyond simply what and how much. We can also employ the software reliability models to give us estimates as to how much more testing is required to meet objectives, etc.

Thus the automatic test plan generation primarily entails two types/phases. One is the use of the unit testing techniques, and the other involves the use of the various software reliability models. System testing plans can be drawn up very early in the life cycle. Many contracts require test plans to be drawn up early in the feasibility phase. This chapter will addresses the automatic generation of the test plan in the two phases separately. The software reliability models will be covered first since their use has been proposed throughout the life cycle. The forecasting and use of unit testing occurs primarily up until the end of the Unit/Unit Testing phase, thus it will be covered last.

8.2 Reliability Models

The various software reliability models provide answers to many possible questions. The questions may deal with the present level of reliability, how much more testing to meet original goals, etc. The effort to perform these calculations can become tedious, especially when attempting a "what-if" type of consultation. Also, it is not always possible to determine what information (i.e. initial data requirements) is available, along with the confidence level of the data.

One of the many goals of a robust KBQA is to automate this process. Automated test planning currently is immature, and worse, does not support requirements-specific considerations. The best possible explanation for this is due to the fact that reliability engineering has not been addressed as a life cycle process, and has relied only on initial specification and final measurement. Since the KBQA is a life cycle process, it should be possible to develop automatic requirements-specific test plan generation.

The most frequently used and best understood of the software reliability models is the Musa Execution model. There are two primary forms of the models:

1. **Basic** Execution Model - linear shape.
2. **Logarithmic** Poisson Execution Model - nonlinear shape.

Both models are nonhomogeneous, in that the characteristics of the probability *distributions that make up the random process* vary with time (variation of failure intensity with time). The models differ slightly in the shape of the curves, and the fact that a Mean Time To Failure (MTTF) exists for the LP model, but not the Basic model. There are two parameters for these models, the execution time and the calendar time. The correct sequence is to choose the execution time, then use the result for the calendar time.

The discussion of automatic test plan generation in this chapter will be based only the Basic Execution Model for the execution parameter. A pictorial description of the model is shown in figure 8-1. The information in the Failure Data box must be either predicted or estimated by use of the KBQA. The choice of or combination of prediction and estimation is primarily determined by the stage of the life cycle. The database like icon (the barrow) is information that is normally specified, however, the KBQA should have the capability to "recommend" values based upon the current circumstances and intelligent decisions from the knowledge base. Also, automated "what-if" sessions cou'd be conducted with a specified/Knowledge Base-derived range. The methodology proposed is extensible to the other models.

Musa BASIC Execution Time Model (Execution Time Only)

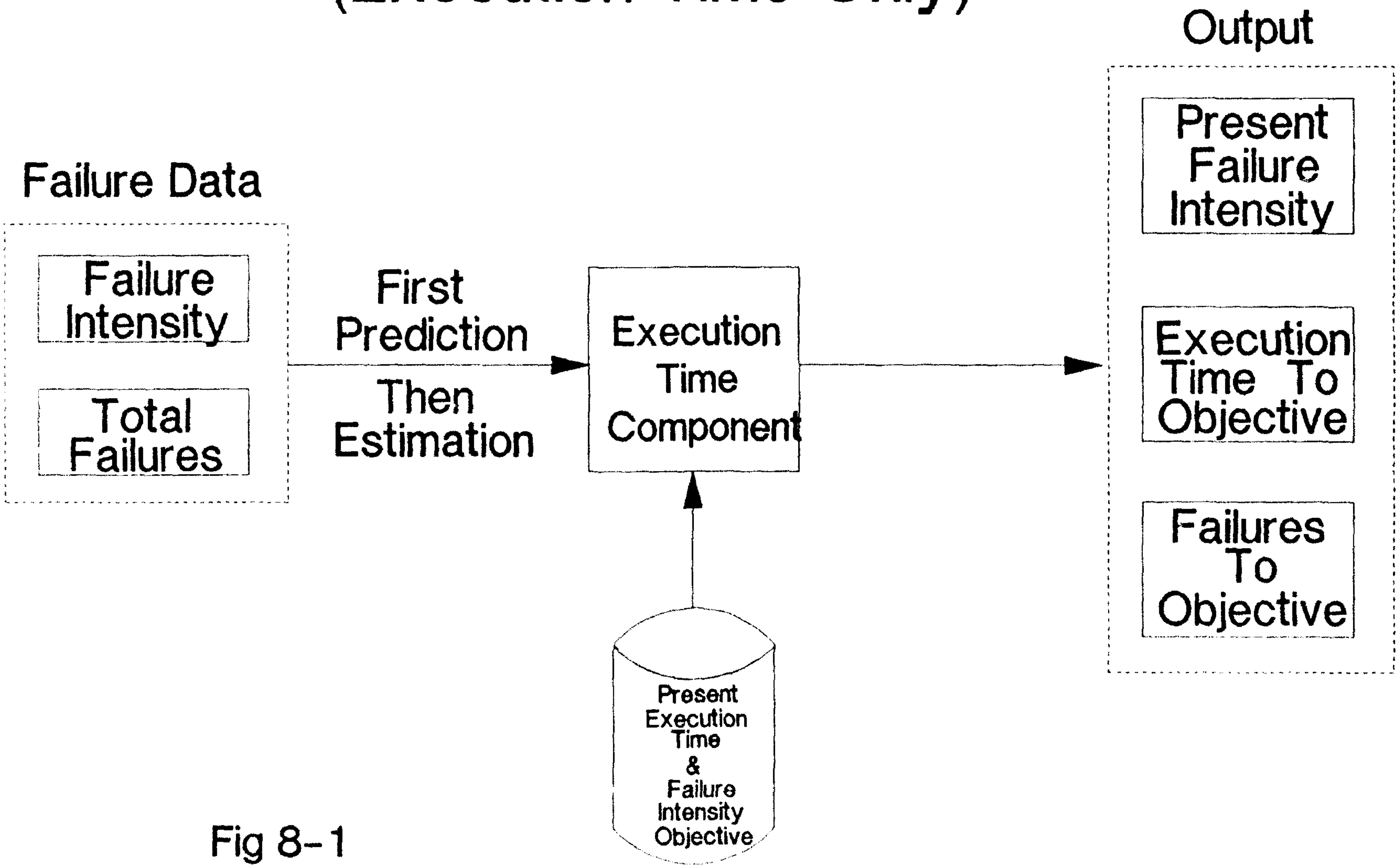


Fig 8-1

8.3 Musa Example

The Musa Basic Execution Model for the execution component model can provide several quantities related to a program's execution time. The formulas require that two parameters be known:

- $Lambda_0$: Failure Intensity at the start of operation.
- v_0: Total number of failures that will ever be experienced during infinitely long operation.

The determination of the values of these components may be performed by two means:

1. **Prediction**: The parameters are prediction from characteristics of the software itself, the development process, etc.

2. **Estimation** - The parameters are estimated during phases of the life cycle where part or all of the software is executed. Upon generation of failure data, the parameters can be estimated statistically.

Prediction of the parameters can be accomplished before any execution of the software is performed. Specifically, Musa defines the Total Estimated Failures to be equal to the Number of Inherent Faults divided by the Fault Reduction Factor. However, the most straight-forward metric for the number of faults is based upon a combination of the software size and complexity, tuned by prior experience with similar projects and the software development process in place.

There are commercially available software applications which perform analysis of the code, and automatically generate metrics based upon size, complexity, and other data. The KBQA should not attempt to duplicate this functionality already procurable. What the KBQA should attempt to do in the prediction of the number of faults is develop a capability to assist in prediction the reliability based upon such factors as:

- Prior experience with similar projects.
- Change: Specifications, Personnel, development Environment, etc.
- Maturity: Software Development Process, Personnel skill and eduction.

- Thoroughness of documentation throughout the life cycle.

It would be desirable for the KBQA to have the knowledge to merge together metrics of the code itself, with the knowledge of the aforementioned factors. In addition, the KBQA should have a feedback loop to continually adjust it's knowledge of these factors to improve it's prediction accuracy.

Once the software is executed and failure data is available, the parameters can be estimated statistically from the data. This estimation process can occur at several levels: Code/Unit Testing, Integration/Integration Testing, System Testing, and Operational and Maintenance phases. Of course, the accuracy of the estimation improves in the later stages of the life cycle. The KBQA should have the knowledge to make the transition from pure prediction, to partial prediction and partial estimation, to total estimation. In this way, these estimations are incrementally used to refine the values established by prediction. As the sample size increases, more emphasis should be placed upon estimation. This blend from prediction to estimation can be based upon the weighted average of previous parameter predictions along with parameter estimations (with the weights being a function of sample size). It may be best if the weights for the estimates be kept at zero until the estimates have an accuracy within an order of magnitude of that of the predictions. The weights for the predictions would be set to zero when the estimates become superior in accuracy by an order of magnitude. Of course, the cut-off point does not have to be a factor of ten, but may be picked on the fly by the KBQA.

Musa also defines the Initial Failure Intensity to be a product of the Total Program Execution Rate, Total Number of Faults in the Program, and the Fault Exposure Ratio. The same processes above can be used by the KBQA to assist in the prediction and estimation of this parameter. In general, there is sufficient means available to allow the KBQA to arrive at intelligent decisions for the process of parameter prediction and estimation. Of course, once in the Operations and maintenance phase, the parameters can be measured and kept within the KBQA database.

8.4 Unit Testing

In the discussion of high quality software, the discussion of unit testing seems more of a contradiction. The very fact that the software is being tested implies that errors are expected. However, testing does have to be performed. Hopefully, the fault density will be less using the KBQA than without it.

Meyers defines seven basic types of software tests. The tests, by no coincidence, are related to the life cycle. The tests are:

- Unit Testing.
- Integration Testing.
- External Function Testing.
- Regression Testing.
- System Testing.
- Acceptance Testing.
- Installation Testing.

The KBQA must address all of these test types in one form or another. This project will focus on the discussion of Unit Testing and Integration Testing as a means to increase reliability.

The two primary types of such tests are functional and structural, and a third has been defined by combination of the two and is called hybrid tests. In figure 8-2, is a diagram of the proposed object hierarchy for the "Test Type" class. At one end of the hierarchy are the various types of tests, while at the other end are the parameters involved with each kind. Notice that this arrangement parallels the earlier discussion with the software reliability models hierarchy. The forward or backward inferencing depend upon two factors:

- What information is available (along with the confidence level).
- What information is desired.

The system would thus inference forward if given certain input data to yield a recommend test type. Conversely, it would inference backward if the environment forced a specific test type. Of course, give uncertain data on both ends, the system would

TEST TYPE HIERARCHY

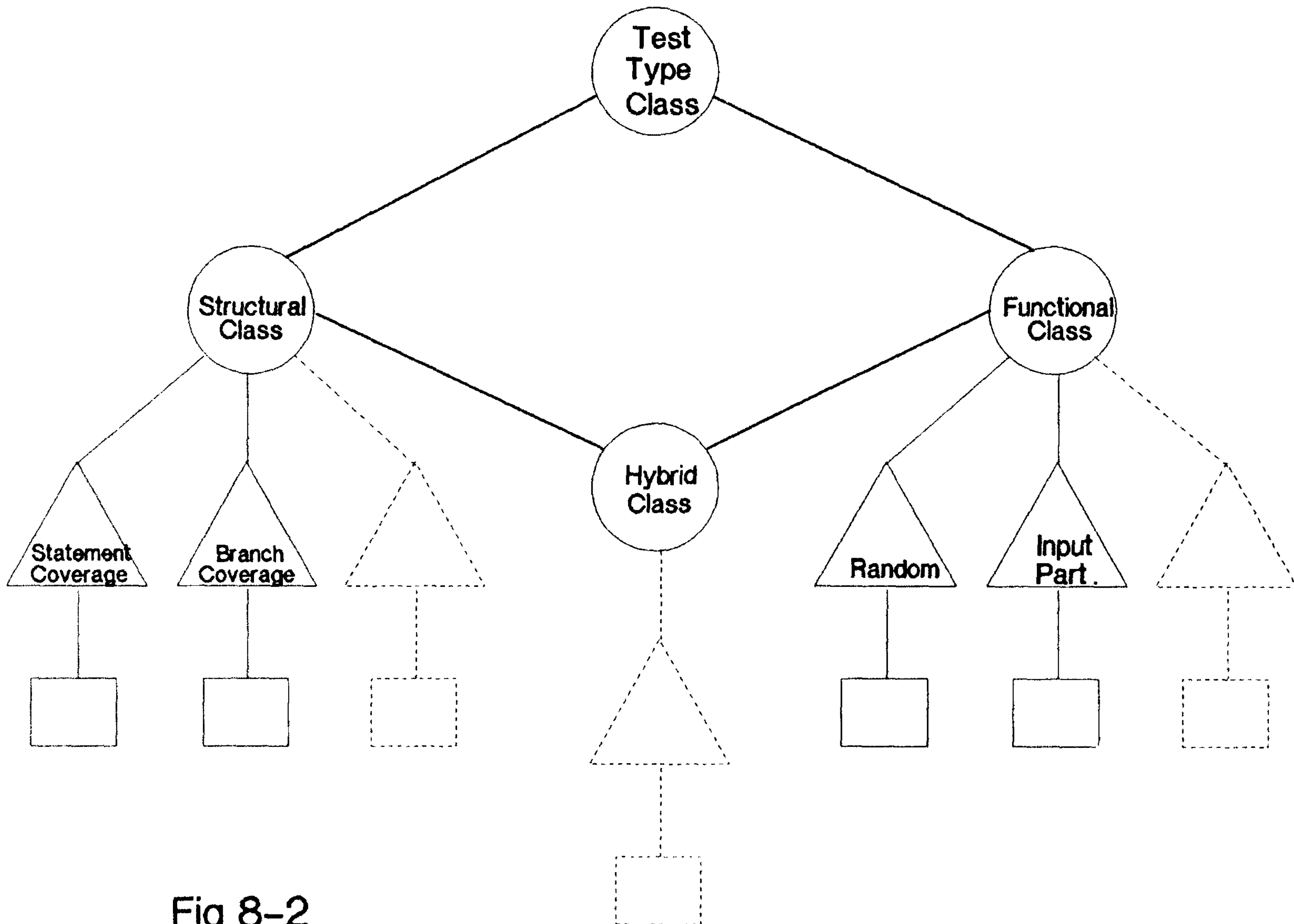


Fig 8–2

inference both ways searching for an optimized solution yielding the highest confidence level.

There are several difficult issues that the knowledge base must be capable of addressing. For structural tests (white box), a major goal is to test smartly and not exhaustively. The chances of achieving complete coverage in most modules is impossible, if not impractical. The knowledge base must be able to determine what paths and with which input values are required. The domain for this is probably based upon the kind of function being implemented (i.e. mathematical function versus logical decisions). It is normally impossible to test every possible value for math functions (i.e. a square root function), but definitely possible to test a logical IF-THEN-ELSE or CASE statement (provided the level of nesting is shallow).

For functional tests (black box), the first issue is input domain partitioning. Automatic generation of this is doubtful, unless the code was generated by extremely exact requirements (i.e. formal methods). Human assisted input domain partitioning may be necessary, but, assisted boundaries of the domain, may be possible. A knowledge base understanding different partitions (i.e. the square root function again) may be possible to develop.

Without a more formal method of software development, it may not be cost effective to attempt to automatically generate test plans. The effort may better be spent on test analysis and reporting. Hopefully, this "lessons learned" information can be fed back into the software development process to find out just what is going wrong.

CHAPTER NINE

Monitoring & Traceability

CHAPTER NINE - MONITORING & TRACEABILITY

9.1 Introduction

An integral part of software reliability engineering is the requirements-specific monitoring of the progress in meeting the defined goals. System reliability goals are normally established in the earlier phases of the software development life cycle. Milestones are then defined throughout the development for the evaluation of progress of meeting those goals.

Another integral part of software reliability engineering is the development of requirements-specific traceability criteria. To properly adhere to DOD-STD-2167A, all documentation and code must have the capability to be traced back to the initial requirements. This is to ensure that all the features required of the system, as defined by the requirements specification are implemented, and that no additional features ("It's not a bug, it's a feature") are put in.

9.2 Software Quality Schematic

The Knowledge Based software Quality Assistant (KBQA) must be designed to facilitate requirements-specific monitoring and traceability. In itself, the KBQA is not the desired mechanism to support such activity. The KBQA is primarily intended to enhance software quality engineering by tracking the progress by prediction, estimation, and measurement. The process of monitoring and traceability is better handled by techniques specialized for software development process modelling.

The comprehensive schematic which, as envisioned, that would be desirable to be employed for software development is shown in figure 9-1. It is anticipated that these "tools" will be utilized for optimizing the development of software. The first tool is a high performance project management tool. A software specific one is not necessary since existing examples do not appear to be promising. A good, general purpose project management tool will integrate smoothly with the existing traditional Management Information System of the parent organization. The primary goals of the project management tool in this schematic is the typical attention to the cost, schedule, and

COMPHRENSIVE SOFTWARE QUALITY SCHEMATIC TO SUPPORT REQUIREMENTS-SPECIFIC TRACEABILITY & MONITORING

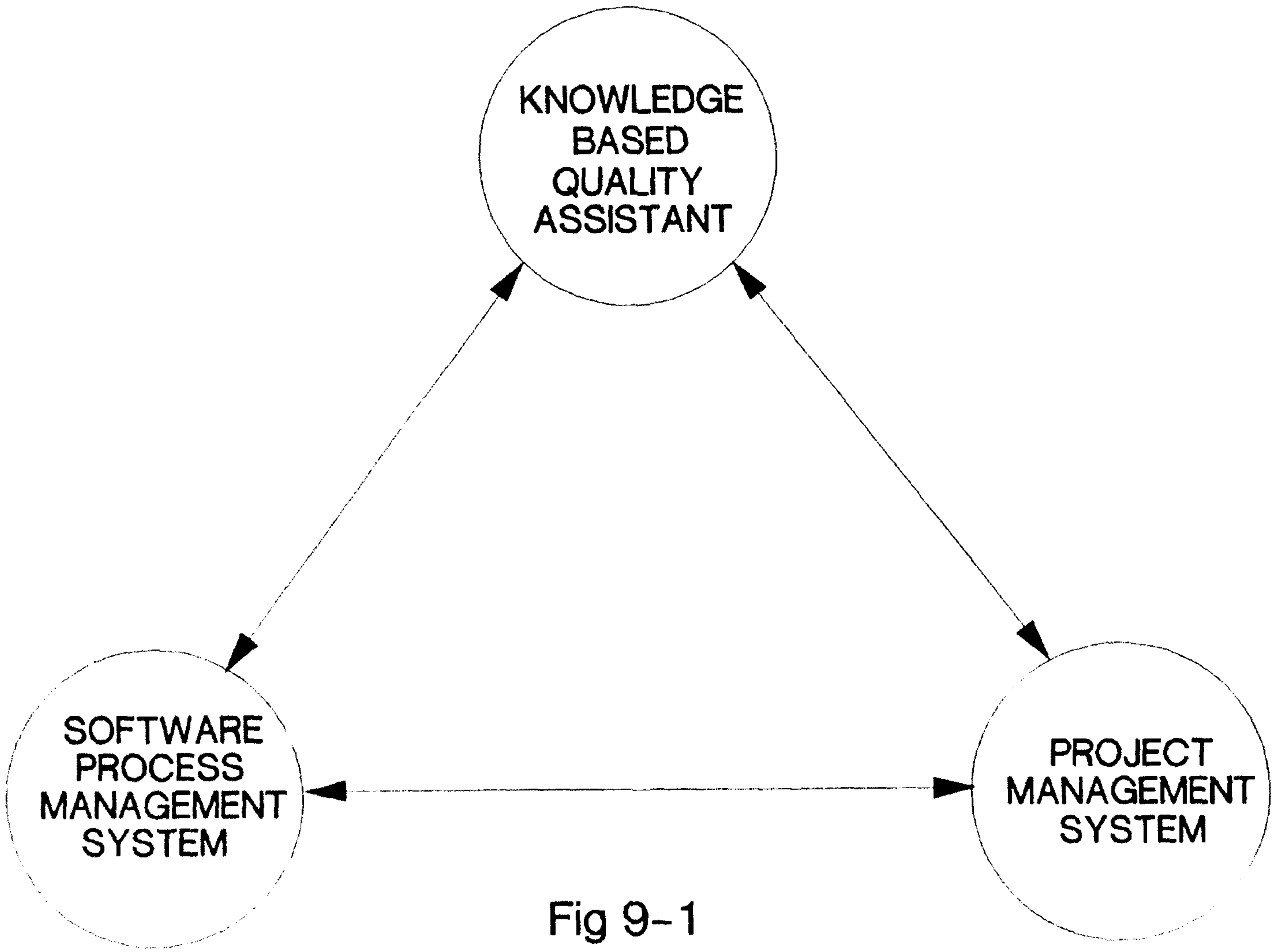


Fig 9-1

performance triangle. Use of a commercial off the shelf project management tool would save considerable duplication of effort in developing a similar functionality within the KBQA.

The second tool envisioned for this software development architecture is a software process management system. Such systems are presently undergoing vigorous research at organizations like IBM, SAIC, and Lockheed. A prototype system will soon be available in the DOD sponsored STARS repository. These systems take into account the Software Engineering Institute Capability Maturity Measurement (SEI/CMM) methodology. The purpose of these systems are to furnish a software-specific management potential to project management. Here too, it is not recommended to duplicate effort in providing the KBQA with similar functionality.

In summary, both Project Management and Software Process Management systems provide the capability to supply the requirements-specific monitoring and traceability of software quality. It is not recommended that this duplication of effort be performed in the KBQA.

9.3 Monitoring & Traceability Issues

The monitoring and traceability of progress throughout the software development life cycle is normally considered to be an exercise in change control. Humphrey [HUMP90] identifies several items which must be maintained (i.e. changed and updated):

- Operational Concept documents.
- Requirements Document.
- Specifications Documents.
- Design Documents.
- Source Code.
- Object Code.
- Test Plans.
- Test Cases, Test Configurations, and Test Results.
- Maintenance and Development Tools.
- User Manuals.
- Maintenance Manuals.
- Interface Control Documents.

This project will not delve into the process of handling such change control. This capability already exists in the software process management schemes. It is believed that creating this capability for the KBQA will be a duplication of effort.

Should some capability of monitoring be desired in the KBQA, it should be possible to develop a rudimentary knowledge base to support such. The approach to the monitoring issues can be based on two strategies:

1. Monitoring with respect to quantitative targets.
    - Setting quantitative targets.
    - Measuring actual values.
2. Identifying components with unusual metric values.
    - Comparing actual values with targets.
    - Responding to significant deviations.

Such rudimentary capability of the monitoring and tracking of requirements can also be enhanced by an automated data analysis system, combined with statistical trend analysis and exception analysis. A knowledge base could also be developed to support report generation and presentation, along with automated conclusion and recommendations.

Activities missing from this approach area risk assessment, cost modelling, test and the general integration of cost, quality, and risk factors. This capability is better served by the aforementioned software process management systems.

CHAPTER TEN

Conclusion and Recommendations

CHAPTER 10 - CONCLUSION AND RECOMMENDATIONS

The reliability of software systems will continue to grow in importance. Today's common acceptance of the "shrink-wrapped" warranty for software will eventually be replaced by regulations and lawyers. As computers continue to be integrated with more and more aspects of life, the consequences of failure will continue to expand. The most efficient and practical means of increasing the reliability of software systems is to place more emphasis on the process, and less on final measurement.

10.1 Conclusions

The concept of a Knowledge Based software Quality Assistant (KBQA) to facilitate the development of quality software is a novel approach. The limited time available (six weeks) constrained the scope of this project to the reliability factor of software quality. The proposed architecture, however, will allow the other software quality factors to incrementally be added in future research. The proposed conceptual architecture of the KBQA is shown in figure 10-1.

Most existing software quality philosophies are based upon first specifying quality, then evaluating it after the fact. This project has proposed that the solution to creating software of higher quality is founded on the software development process. Attention must be paid to software quality throughout the development, and even extending into the Operational and Maintenance phases in an effort to utilize "lessons learned". Software quality must be appraised by a Predict-Estimate-Measure cycle, in which a smooth transition blends from one technique to the next.

The KBQA, as developed in this project will support software reliability engineering. A key goal of the KBQA and it's modular architecture is to make it extensible. The extension of the KBQA as envisioned would support the entire software quality spectrum. Use of the KBQA with an automated Software Process Management System (such as the SEI/CMM or the Krasner/SAIC) and a Project Management System (such as Microsoft Project) would provide a requirements-specific monitoring and requirements-specific traceability capability.

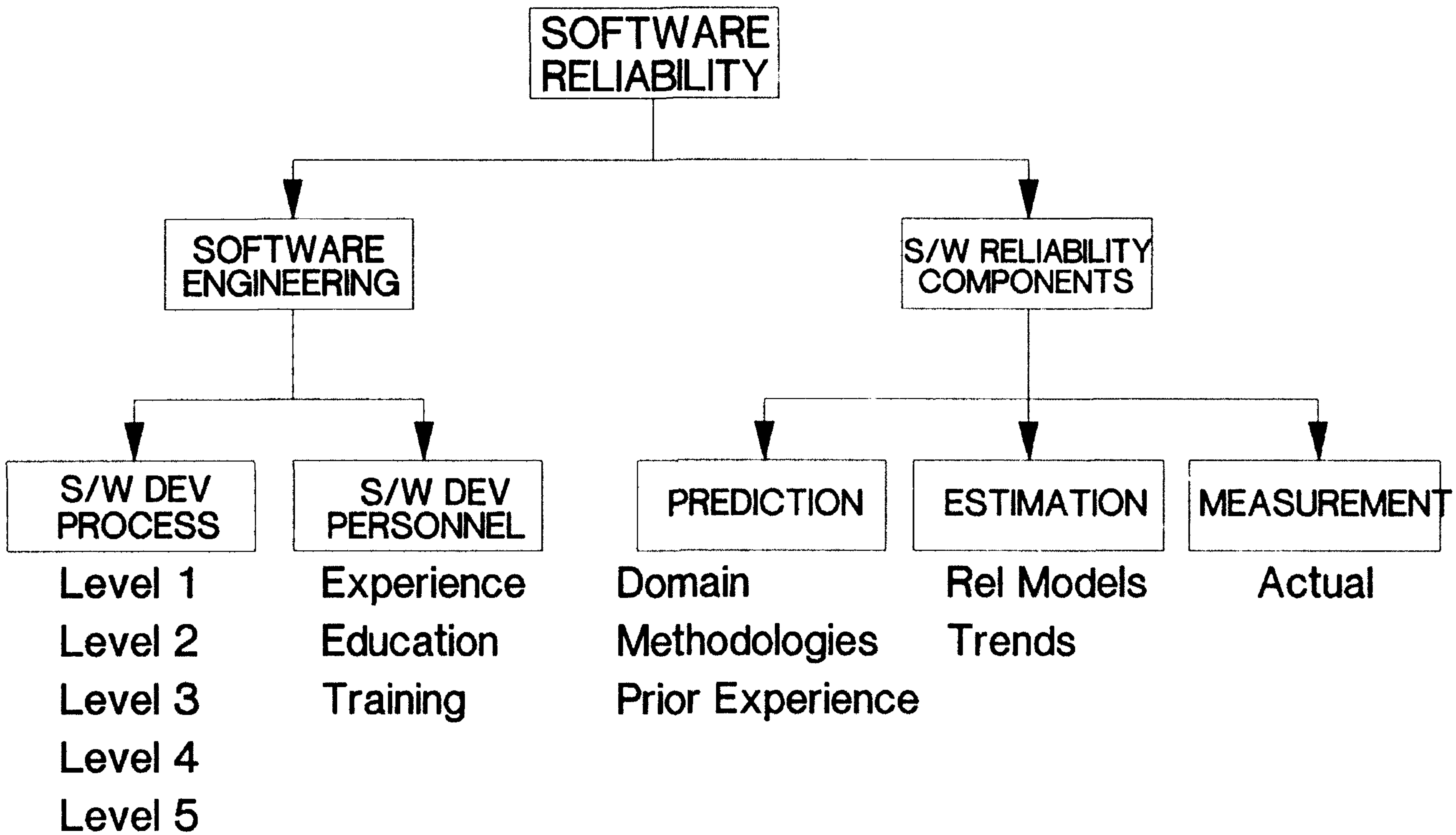

SOFTWARE RELIABILITY
SOFTWARE ENGINEERING
S/W RELIABILITY COMPONENTS
S/W DEV PROCESS
S/W DEV PERSONNEL
PREDICTION
ESTIMATION
MEASUREMENT
Level 1
Level 2
Level 3
Level 4
Level 5
Experience
Education
Training
Domain
Methodologies
Prior Experience
Rel Models
Trends
Actual

Fig 10-1

10.2 Recommendations: KBQA Development Sequence

The KBQA can be developed in a number of different stages. The modular knowledge base architecture would allow the creation of knowledge islands by individual experts in each area. This approach also allows a parallel development of knowledge islands. The following is a recommended succession of events in creating the KBQA:

1. Settling on a hardware and software suite to host the KBQA. Several system combinations are possible, but a key to the KBQA distribution and wide spread use is to host the system on equipment that is readily available. The preferred system is the MacIntosh II Color computer system due to the power, flexibility, and the highly respected ease of use of that computer. The software choices are commercial off the shelf packages; Nexpert Object for knowledge base development, Oracle for the database, Microsoft Project for the project management system, HyperCard for the Man Machine Interface, and Ada as the language of choice for any code that must be written. A difficult choice is between which Software Process Management System to employ (SEI versus SAIC).

2. Next, is to decide on a particular knowledge base architecture from the three provided (or a modification or combination of the three). The knowledge bases can be developed concurrently by different developers.

3. The next task is the detailed design of various object hierarchies. One is the categorization of the various software reliability models. The knowledge base must partition the models based upon required parameters, provided information, etc. Another hierarchy is the various testing methodologies (structural, functional, and hybrid). It is anticipated that many kinds of such hierarchies must be developed to allow efficient run time operation of expert system.

4. An intermediate task is to develop a smooth integration of the work developed in tasks two and three. This is more of a knowledge base refinement and iteration task. Special care must be taken to ensure that "hooks" are in place for the other software quality factors.

5. The final task would be to connect the project management system and a software process management system with the KBQA. The KBQA should be linked with the project management and SPMS systems to support the requirements-specific monitoring and traceability of software reliability.

After the system has been developed to a point where everything is integrated and functioning smoothly, expansion beyond just reliability should be attempted. The next most important software quality factor, just behind reliability, could arguably be maintainability. It is generally excepted that reliability and maintainability are about the only (or at lest most) quantifiable of the thirteen software quality factors. Finally, the remainder of the software quality factors should be introduced within the KBQA, preferably one at a time.

APPENDIX

References

REFERENCES

[MUSA87] Musa, John D.; Iannino, I.; Okumoto, K. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill Series in Software Engineering and Technology. 1987.

[HUMP90] Humphrey, Watts S. *Managing the Software Process.* Addison-Wesley Publishing. 1990.

[IEEE91] *International Symposium on Software Reliability Engineering.* IEEE Computer Society Press. 1991.

[LASK89] Lasky, Jeffery A.; Kaminsky, Alan R.; Boaz, Wade. *Software Quality Measurement Methodology Enhancements Study Results.* RADC-TR-89-317. 1989.

NOTICE: Numerous *Rome Laboratory Technical Reports* that were published over the past decade and a half were studied extensively in an effort to further my understanding of software reliability. No specific references can be provide, yet the material proved invaluable in getting a "feel" and understanding of the state of the art in software reliability engineering. The most helpful of the Technical Reports was RADC-TR-87-181, Volume I and II.

*U.S. GOVERNMENT PRINTING OFFICE: 1993-710-093-60226

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.